uc3m | Universidad Carlos III de Madrid

Grado en ingeniería Informática

2017-2018

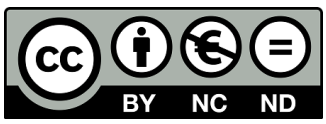
Development of a Search Algorithms Library in Go using DOT language

---

Krzysztof Piotr Statkiewicz

Carlos Linares López

Leganés, 18 de Octubre de 2018



Abstract

The *Go* programming language, also known as *Golang*, is a powerful general purpose language used in high-performance concurrent tasks with a growing community of developers and open source contributors, known mostly for its unique parallel programming paradigm using `channels`. The language has a reputation for heavily opinionated architectural decisions and therefore inclusions and omissions in the specification. Notably, most established data structures included in traditional programming languages are purposefully excluded from it. Part of the reason for it is the lack of *generics*: programmers are encouraged instead to generate structures based on `interface` definitions and fulfill their needs per-implementation.

The project of the thesis is hereby presented, with the following goals: providing a robust and flexible search algorithms library, contributing to the *Go* community with an open source library, and employing the state-of-the-art in search and heuristic search theory and underlying data structure optimizations, with the peculiarity of the inclusion of a module which enables the modeling and solving of problems using `dot` [1] files as input.

The work details the entire process: formal specification, design, planning, development and testing, detailing justifications for any decisions taken and also analyzing the performance of several data structures along the way. Finally, the Appendix contains additional resources and project documentation, generated via `godoc` [2] (Appendix B).

Keywords: open source, *Go*, search algorithms, *dot* language, parser, data structures, interfaces.

Acknowledgements

I would like to give a big thank you to my thesis supervisor, Carlos Linares López, for his suggestion regarding this project and all the materials provided, and his trust in my own judgement during its development. To my parents: no words can express my gratitude towards you and your dedication to provide the best possible life for me. Additional thanks to all my friends and specially my girlfriend Olivia for being so supportive and encouraging; in days where I couldn't see the end you showed me the light.

Dedicated to my University classmates and friends; thank you for enabling us to learn and explore Computer Science together, above and beyond what was provided in lecture halls.

Table of contents

1 Introduction & State Of The Art 3

1.1 About the project 6

2 Analysis, Design and Planning 7

2.1 Requirements 7

2.2 Use Cases 9

2.3 Test Cases 11

2.4 Design Proposal 13

2.5 Traceability matrix 17

2.6 Planning 18

2.7 Management systems 19

2.7.1 Version Control 19

2.7.2 Testing and Continuous Integration 19

2.7.3 Project management 19

3 `gost`: Go Data Structures Library 20

3.1 `NodeList`: a single-linked sequential container 21

3.2 Queues 22

3.2.1 type `Queue` 22

3.2.2 type `NodeQueue` 22

3.2.3 Benchmarks and comparison 22

3.2.4 type `PriorityQueue` and `MinPriorityQueue` 25

3.3 Stacks 26

3.3.1 type `Stack` 26

3.3.2 type `NodeStack` 26

3.3.3 Benchmarks and comparison 26

4 `search`: Search Algorithms Library 27

4.1 `interfaces` defined by the package 28
4.2 Algorithms Preface 28
4.3 Best-First algorithms helper 30
4.4 Blind Algorithms 32
4.4.1 func `BreadthFirst` 32
4.4.2 func `DepthFirst` 33
4.4.3 func `DepthFirstBranchAndBound` 34
4.4.4 func `Djikstra` 35
4.4.5 func `IterativeDeepening` 35
4.5 Informed Algorithms 37
4.5.1 func `AStar` 37
4.5.2 func `Beam` 37
4.5.3 func `GreedyBestFirst` 38
4.5.4 func `HillClimbing` 38
4.5.5 func `IterativeDeepeningAStar` 38

5 `dot`: *Dot* File Parser 39
5.1 type `Graph` 39
5.2 CLI Program 40
5.3 Parsing helpers 41
5.4 Parser 42

6 Applicable regulations 43

7 Socioeconomic Context & Impact 45

8 Project Budget 46

9 Final Remarks 47
9.1 Future Work 48

Bibliography 49

List of Figures

1.1 Worldwide trend analysis for keywords “search algorithm” and “deep learning” between 2013 and 2018 4

1.2 TIOBE index for Go programming language [9] 5

2.1 *Type Diagram* for package `dot` 14

2.2 *Type Diagram* for package `gost` 15

2.3 *Type Diagram* for package `search` 16

2.4 Project Timeline (Gantt chart) 18

3.1 Example of `gobench` output 23

3.2 Comparison between `Queue` and `NodeQueue` 24

3.3 Example of *heap* structure 25

3.4 Comparison between `Stack` and `NodeStack` 26

4.1 *Type Diagram* of the interfaces exposed by `search` 28

4.2 Pseudocode for Best First helper 31

4.3 Pseudocode for BFS algorithm 32

4.4 Pseudocode for DFS algorithm 33

4.5 DFS Branch And Bound pseudocode from *Artificial Intelligence: Foundations of Computational Agents* [21] 34

4.6 IDS pseudocode from *Artificial Intelligence: Foundations of Computational Agents* [21] 36

5.1 Token definitions in *Go* 41

5.2 Parser pseudocode 42

List of Tables

1 Document Change History 1

2.1 Listing of Functional Requirements 7
2.2 Listing of Functional Requirements (Continued) 8
2.3 Listing of Non-functional Requirements 9
2.4 Use Case UC-01 9
2.5 Use Case UC-02 9
2.6 Use Case UC-03 10
2.7 Use Case UC-04 10
2.8 Use Case UC-05 10
2.9 Use Case UC-06 10
2.10 Use Case UC-07 10
2.11 Test Case TC-01 11
2.12 Test Case TC-02 11
2.13 Test Case TC-03 11
2.14 Test Case TC-04 11
2.15 Test Case TC-05 12
2.16 Test Case TC-06 12
2.17 Test Case TC-07 12
2.18 Test Case TC-08 12
2.19 Traceability Matrix 17

3.1 `gost` data structures 20

4.1 `search` algoritm collection 27

8.1 Project associated costs 46

Version	Date	Description
1.0.0	12/05/2018	Initial release
1.0.1	14/05/2018	Simplified Best-First algorithms into common function explanation
1.0.2	18/05/2018	Typographic errors and style changes
1.0.3	02/06/2018	Added “snippet” command for boxed code snippets
1.0.4	07/07/2018	Restructuring of gost section, benchmarks and python script appendix
1.1.0	26/07/2018	Arquitechtural redesign; removed search.Domain interface in favor of State interfaces. New type diagrams, rewrite of `dot` chapter
1.1.1	29/07/2018	Arquitechtural redesign; Rewrite of `search` chapter to conform to new functions structure
1.1.2	30/07/2018	Added interface diagram in `gost` chapter. Added conventions section for algorithms; updated blind algorithm sections
1.1.3	31/07/2018	Updated informed algorithm sections
1.1.4	01/08/2018	Typographic errors revision; redone benchmark charts
1.1.5	25/08/2018	Added explanatory algorithm figures and code example of `gost` token
1.1.6	26/08/2018	Miscleaneous typographic errors fixed; minor text changes
1.1.7	01/09/2018	Corrected `gost` type diagram
1.1.8	03/09/2018	Updated `search` algorithms sections; reverted “snippet” to monospace only
1.1.9	04/09/2018	Fixed reference issues and regulations chapter
1.2.0	09/09/2018	Fixed typos and minor changes
1.2.1	24/09/2018	Thesis defense date and location set

Table 1. Document Change History

1. Introduction & State Of The Art

Ever since the creation of household computers, society has undercome a deep transformation on nearly every possible level of human activity. From social interaction, learning and information retrieval, to business processes and public services; all of them have adapted and restructured to accept the digital era of information. Moreover, with the recent increased interest within the computer science community towards Artificial Intelligence and Machine Learning, which in turn is already transforming both mundane things like household devices or cars and complex, advanced business solutions, the reality is that software plays a central role in society.

Sundar Pichai, CEO of Google, stated at the end of 2016: “The last 10 years have been about building a world that is mobile first. In the next 10 years, we will shift to a world that is AI first” [3]. The last five years alone have seen a massive increase in interest and effort centered towards AI, specifically regarding Machine Learning research and development. IDC forecasted at the end of 2017 a 50.1% compound annual growth rate for global expenditures on AI research and development, making a total of 57.6 billion US Dollars spent by 2021 [4].

Despite this recent surge of interest in the aforementioned field, it is important to remark that the underlying principles and research they are based on aren’t actually recent breakthroughs. For instance, we can trace back the history of neural computing to as early as the 1940s with McCulloch and Pitts’ paper - *A logical calculus of the ideas immanent in nervous activity* [5]. The research community at the time achieved some steps but eventually abandoned the concept for part of the century, eventually slowly regaining interest and leading to important breakthroughs such as convolutional neural networks and *Deep Learning*, with the success in 2009 of the *ImageNet* image classifier [6] being considered as the spark for the currently renewed interest.

While the spike in Machine Learning related research is very promising in the light of AI breakthroughs, it also raises some wariness that efforts may shift over from other fields of research. In parallel to neural network advancements during the last century, a separate, more systematic and mathematical approach at problem-solving was also researched: State space search and optimization. Led by researchers such as Judea Pearl, Rina Dechter and Richard Korf, this alternative AI focus centered around around topics such as probability, combinatorics and the mathematical notion of the *Solution Space*, leading to such widespread models as Bayesian Networks. Many of those research findings have found their way into powering some of the solutions that are consumed on a daily basis by millions of people worldwide, ranging from robotics to planning, optimization, and computer games. In fact, chances are that the

reader has recently interacted with a problem solver developed using a search algorithm: for example, the optimal route traced in a smartphone to traverse an unknown location or choose the shortest work commute, to an indirect impact due to solvers of very complex heuristic and optimization problems in the enterprise.

Despite this, there is a discernible difference in awareness and common knowledge about this field of research in contrast to Machine Learning. A simple search on the Google analytics platform shows the leap on a global scale between both fields throughout the last five years (fig. 1.1) [7]:

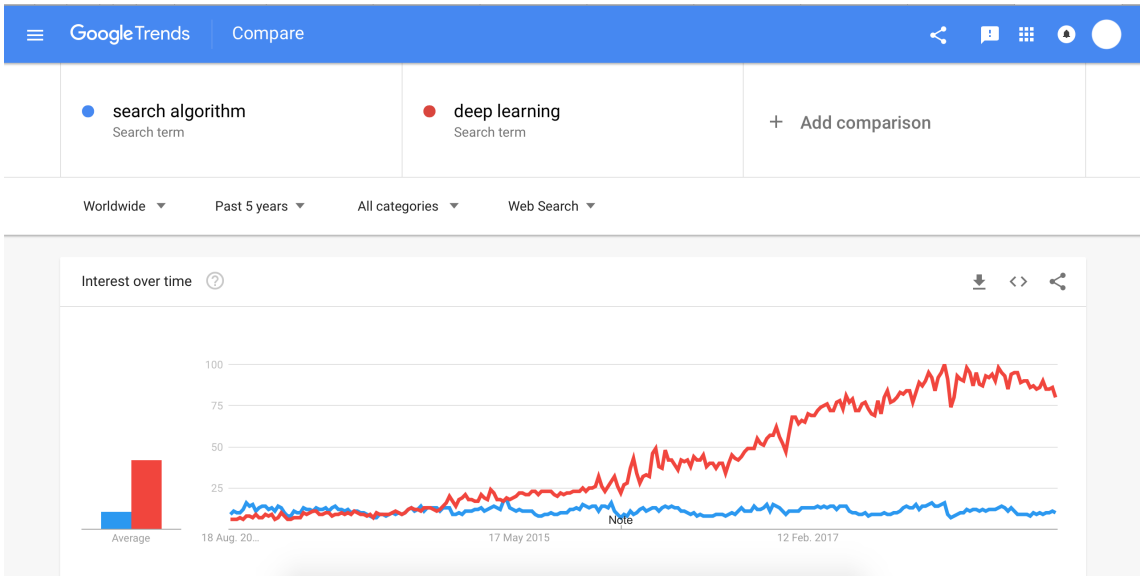


Fig. 1.1. Worldwide trend analysis for keywords “search algorithm” and “deep learning” between 2013 and 2018

Furthermore, another notable difference in the development community is the lack of mature and standardized frameworks. Part of the fault in this may be the drastic difference in the nature of the solutions: neural networks can be thought of as highly configurable, complex software on its own. Further specializations can also apply based on their purpose, which has led to a tight set of highly polished “off-the-shelf” implementations: frameworks such as *Tensorflow*, *Caffe* or *Keras* are commonly considered standard within the community.

On the other hand, search algorithms, perhaps in part due to their relatively low implementation complexity and due to the astounding amount and variety of general-purpose programming languages, picture a much more scattered landscape. Honorable mentions can be given to *Aima*, a search algorithms library developed for several languages like Java, Python and C# as a companion for the book “Artificial Intelligence - A Modern Approach” [8] by Stuart Russell and Peter Norvig as well as the C++ *Boost* library (as found in https://www.boost.org). However, when outside of the realm of these two highly polished libraries and their available language implemen-

tations - for example due to the usage of a newer programming language - it's up to the developers to implement their own crafted solution. This may potentially lead to undesirable side effects, such as incorrect logic within the algorithm (which for instance *Aima* avoids by publicly peer-reviewing their source code). Another consideration is the fact that this generates fragmentation and a rupture of the concept of *DRY* (Don't Repeat Yourself): different development teams will potentially put efforts into implementing the exact same algorithms, because of the lack of an "off-the-shelf" implementation for their production language.

One of such new languages is Google's own *Go*, also known as *Golang*. According to programming language surveys such as TIOBE [9] and Stack Overflow [10], the gopher language has been steadily growing in popularity in recent years, specially in the field of data science, tightly interleaved with Machine Learning.

Fig. 1.2. TIOBE index for Go programming language [9]

However, being a rather new player in the arena of programming language giants like Java and C++ it doesn't have a standardized framework of search algorithms. To fill that gap, a reference search algorithms library implementation is a necessary step towards the establishment of *Go* as a valuable, production-ready language. The aim of this thesis is to provide exactly that: an essential set of algorithms conveniently packaged and tested to use in any context.

The reader might be inclined to think that providing yet another library doesn't seem *enough* as, per definition, it is a language-specific implementation of a set of search algorithms. To address this concern, and moreover, to increase and expand the accesibility of said algorithms to any kind of users regardless of their particular project tools and/or requirements, an additional development proposition is made: the inclusion of `dot` language, as defined per *Graphviz* [1], as an optional input language

for the usage of said search algorithms, effectively enabling anyone to utilize the library under a standardized file format, with the flexibility to customize and implement more advanced solutions depending on the individual needs.

1.1. About the project

The project consists of a search algorithms library, further enhanced with an entity capable of parsing `dot` files representing state spaces. Because of the *Go* language's nature, an additional prerequisite consists of implementing any required underlying data structures for either search algorithms or `dot`-related needs. Hence, the main project is divided into three separate sub-projects, arranged as `packages` (equivalent of modules or libraries in other languages). Let's briefly define each one:

- `search`: a search algorithms library. Exposes several State-related *Go* interfaces as well the algorithms as function which use said interfaces.
- `gost`: a minimal data structures library, focused on optimization and generalization for search algorithms.
- `dot`: a *.dot* [1] file parser, converts the file's problem domain information into a *Go* data structure which implements the aforementioned State-interfaces from `search`.

The packages are named in a succint, self-explanatory way as they are the accessor to everything they expose to the user, following the language's standard [11].

It is worth remarking that the thesis' structure doesn't reflect the development process, as all three sub-projects are tightly coupled and iterative changes affect the final outcome. The development itself is test-driven, and on an iterative basis, following ideas drawn from *agile development* more so than traditional Software Engineering. Of particular note, data structures selection and testing is a core interdependency between both `gost` and `search` which requires testing and benchmarking to select the most appropriate match. For the reader's convenience, all decisions are reflected under the chapter devoted to each of the libraries.

The Thesis is subdivided roughly into the following chapters: Requirements and traceability, Development of each library and decisions carried out, Applicable Regulations, Socioeconomic Context & Impact, Conclusions and finally an Appendix including the documentation that the language generates from the code documentation via the `godoc` [2] command (Appendix B). The chapters regarding Applicable Regulations and Socioeconomic Context & Impact can be found under Chapters 6 and 7, respectively.

2. Analysis, Design and Planning

2.1. Requirements

As explained in the introduction, the goal is to implement a robust search algorithms library which leverages a `dot` parser. However this definition is too broad and ambiguous and doesn't fully elaborate on what is actually required for the project. Hence the need to compile a formal specification of the requirements of the project:

ID	Description	Use Cases	Priority
FR-01	The system shall provide a parser of *.dot* files that converts graph information into a parameterized `type`.	UC-01, UC-02, UC-03, UC-04	High
FR-02	The parser shall be accessible as a standalone command-line program for inspection of *.dot* file correctnes, with an optional argument for listing of the contents of the resulting `type`.	UC-01	Mid
FR-03	The system shall define a set of `interfaces` which allow the user to utilize the search algorithms, provided a `type` that implements the method signatures of one.	UC-02, UC-03, UC-04, UC-05.	High
FR-04	The search algorithms `interfaces` shall provide a method signature to obtain the neighbors of a vertex passed as parameter.	UC-02, UC-03, UC-04, UC-05	High
FR-05	The search algorithms `interfaces` shall provide a method signature to obtain the cost of traversing from a node to a neigboring node.	UC-02, UC-03, UC-04, UC-05.	High
FR-06	The search algorithms `interfaces` shall provide a method signature to obtain a heuristic cost given a node.	UC-02, UC-03, UC-04, UC-05.	High
FR-07	The `type` exposed by the *.dot* parser shall implement the method signatures of the exposed `interfaces`.	UC-02, UC-03, UC-04, UC-05.	High

Table 2.1. Listing of Functional Requirements

ID	Description	Use Cases	Priority
FR-08	The `type` exposed by the *.dot* parser shall offer utility functions for setting and accessing vertex and edge attributes given a graph.	UC-01, UC-02, UC-03, UC-04	Mid
FR-09	The `type` exposed by the *.dot* parser shall store the attributes of every vertex and edge, preserving directionality in edges.	UC-01, UC-02, UC-03, UC-04	High
FR-10	The `type` exposed by the *.dot* parser shall keep attribute accessors to obtain any defined traversal costs between edges and the value of a heuristic in nodes.	UC-01, UC-02, UC-03, UC-04	High
FR-11	The `type` exposed by the *.dot* parser shall allow the user to define a custom cost and heuristic function, effectively overriding the built-in key-value reader.	UC-04	High
FR-12	The system shall provide the user with a set of search algorithms which consume a `type` that implements the aforementioned `interface`.	UC-02, UC-03, UC-04, UC-05, UC-07	High
FR-13	The search algorithms shall optionally provide metadata, namely execution time and number of expanded nodes.	UC-02, UC-03, UC-04, UC-05, UC-07	Low
FR-14	The following search algorithms shall be implemented: BFS, DFS, DFBnB, Djikstra, IDS, A*, Beam Search, GBFS, Hill Climbing, IDA*.	UC-02, UC-03, UC-04, UC-07	High

Table 2.2. Listing of Functional Requirements (Continued)

ID	Description	Use Cases	Priority
OR-01[1]	The source code must comply with the *Go* specification, guidelines, standards and best practices [11].	–	Mid
OR-02	The *.dot* file parser must cover a minimal subset of the language to cover graph descriptions with edge and vertice weights and both unidirectional and bidirectional edges.	UC-01, UC-02, UC-03, UC-04	High
PR-01[1]	The search algorithms shall use data structures with a good generalized performance, since the generic nature of the `interface` does not allow for domain-specific optimizations.	UC-01, UC-02, UC-03, UC-04, UC-05, UC-06	Mid

Table 2.3. Listing of Non-functional Requirements

2.2. Use Cases

The use cases can now be defined. In this project, the main actor will always be the *end user* (*i.e.* developer, user of the library) hence the definitions can be streamlined with tabular entries of IDs and descriptions:

ID	UC-01
Description	User validates a *.dot* file against the command-line program, visualizing the contents of the resulting `type` in the terminal.

Table 2.4. Use Case UC-01

ID	UC-02
Description	User solves an instance of a problem defined as a graph in a *.dot* file by using the provided `type` and search algorithms.

Table 2.5. Use Case UC-02

[1] `PR` stands for *Product Requirement* and `OR` for *Organizational Requirement*, respectively.

ID	UC-03
Description	User solves an instance of a problem defined as a graph in a *.dot* file by using the provided `type`, specifying costs and heuristic values.

Table 2.6. Use Case UC-03

ID	UC-04
Description	User solves an instance of a problem defined as a graph in a *.dot* file by using the provided `type`, overriding the `cost` and `heuristic` function to suit more complex needs.

Table 2.7. Use Case UC-04

ID	UC-05
Description	User implements a `type` fulfilling the `interface`, enabling the usage of the provided search algorithms.

Table 2.8. Use Case UC-05

ID	UC-06
Description	User employs data structures provided for other unspecified purposes.

Table 2.9. Use Case UC-06

ID	UC-07
Description	User employs search algorithms library for educational/research purposes, obtaining execution times and node expansion data.

Table 2.10. Use Case UC-07

2.3. Test Cases

The following test cases are meant to serve as a template for all the possible actions a user may attempt to do. The fine details are unspecified due to unknown implementation specifics; however, development and testing phases shall use the following routinely as a basic building block to trace correct operation and any possible regressions:

ID	TC-01
Description	User launches the CLI command with a file path as a parameter.
Preconditions	The user has imported and built `dot`.
Result	The terminal prints any syntax errors, or an error message if the file is invalid. Additionally, if the verbose option is set the structure and contents of the file are listed.

Table 2.11. Test Case TC-01

ID	TC-02
Description	User parses a file by directly invoking the public functions from `dot`.
Preconditions	The user has imported `dot` and is using valid file paths.
Result	The function shall return a valid `dot.Graph` object with the contents and structure of the graph in the *.dot* file.

Table 2.12. Test Case TC-02

ID	TC-03
Description	User attempts to use a data structure from `gost` for an unspecified purpose.
Preconditions	The library is imported into the User's project.
Result	The user can import all public `gost` types and invoke their methods.

Table 2.13. Test Case TC-03

ID	TC-04
Description	*Umbrella Test Case.*[2] Ensure correct behavior of (every) search algorithm.
Result	The user is able to employ any algorithm to solve the problem.

Table 2.14. Test Case TC-04

ID	TC-05
Description	User attempts to solve a problem defined in a *.dot* file.
Preconditions	The File has been parsed and the resulting `dot.Graph` type is used. No extra configuration is required for the solver.
Result	The user is able to employ any algorithm to solve the problem.

Table 2.15. Test Case TC-05

ID	TC-06
Description	User attempts to solve a problem defined in a *.dot* file setting a *CostKey* and/or *HeuristicKey*.
Preconditions	The File has been parsed and the resulting `dot.Graph` type is used. Either/both fields in the type instance are correct and have been set.
Result	The user is able to employ any algorithm to solve the problem.

Table 2.16. Test Case TC-06

ID	TC-07
Description	User attempts to solve a problem defined in a *.dot* file setting a *CostFunc* and/or *HeuristicFunc*.
Preconditions	The File has been parsed and the resulting `dot.Graph` type is used. Either/both functions in the type instance are correct and have been set.
Result	The user is able to employ any algorithm to solve the problem.

Table 2.17. Test Case TC-07

ID	TC-08
Description	User provides own implementation of any of the `intefaces` to solve a search problem.
Preconditions	The interface is correctly implemented and problem well defined.
Result	The user is able to employ any algorithm to solve the problem.

Table 2.18. Test Case TC-08

[2]The test case can be subdivided into several, similar cases. Splitting it formally doesn't provide further value, however, hence it's the developer's duty to ensure compliance when applicable.

2.4. Design Proposal

With the requisites and use cases specified, the software design/architect can be proceeded with. One of the most standardized tools for this task is the Unified Modeling Language, or *UML* [12]. However, the specification was designed with Object-Oriented languages in mind, which doesn't map accurately to the software architecture. In *Go*, functions are *first-class citizens*. Hence functions should be defined on the same level as Objects (in this case, *Types*) - as entities that interact among themselves.

As *UML* doesn't provide tools that satisfy these needs, a modified version of class diagrams is used instead, named *Type Diagrams* [3]. The reader will find of particular relevance the following adaptations:

- `func` definitions are allowed to relate on the same level as `type` definitions.
- *Go*'s integrated test framework allows to keep test files that are not packaged upon compilation. Any test functions/types/packages will be marked by **boxes with a dashed border**.
- Definitions of types follow a simmilar structure to Classes in *UML*. Functions, however, will be defined by their signature (name, input arguments and output arguments, if any). Public functions begin by uppercase letters, and private functions with lowercase, as that's the language's syntax.
- In *Go*, types can be composed into another. When composing, the composed type's methods are not *"inherited"*, however. For this particular association, the type that composes the *base* type will point a **hollow arrow with dashed lines** and the keyword *"Composes type"*. By default, the association is assumed to be 1-to-1; if the composed type is a collection of the base type, the *composed* type will show next to the arrow an n-arity of * and the *base* type, 1.
- In case of a package interacting with another (other than from the standard library), the external package's box will be displayed, along with any items that may directly interact with the current package.

[3] functions are also a type in *Go*

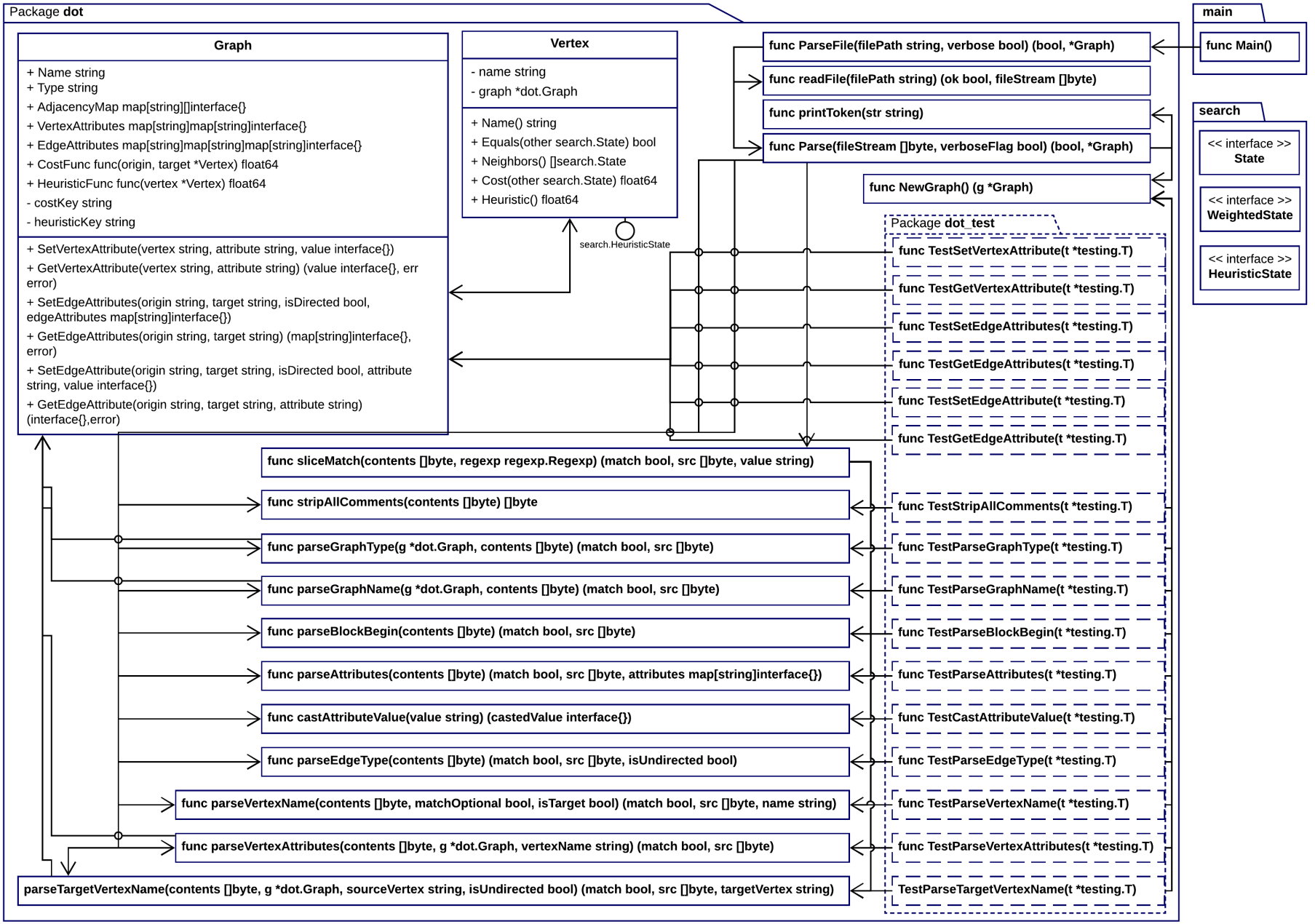


Fig. 2.1. *Type Diagram* for package `dot`

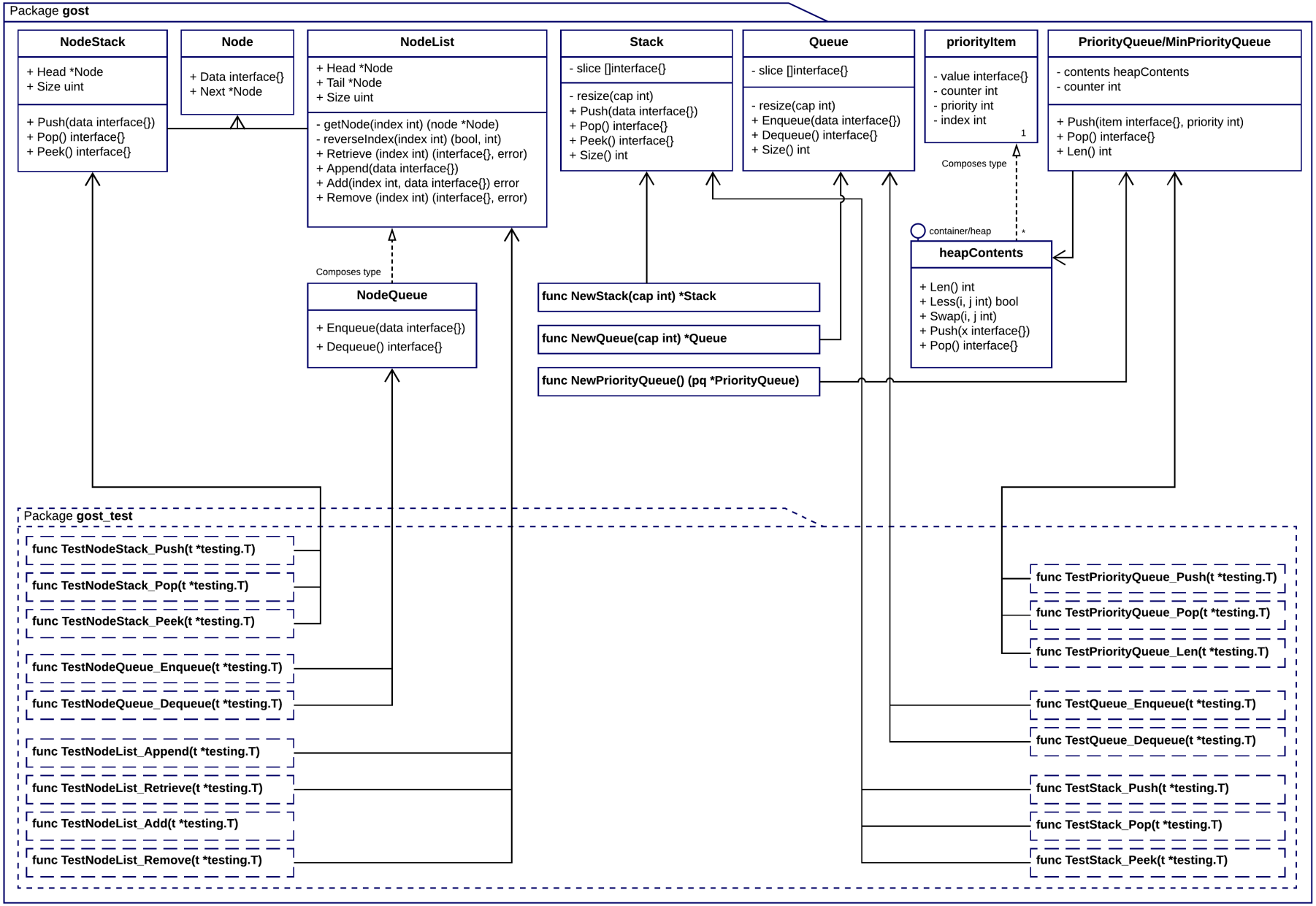


Fig. 2.2. *Type Diagram* for package `gost`

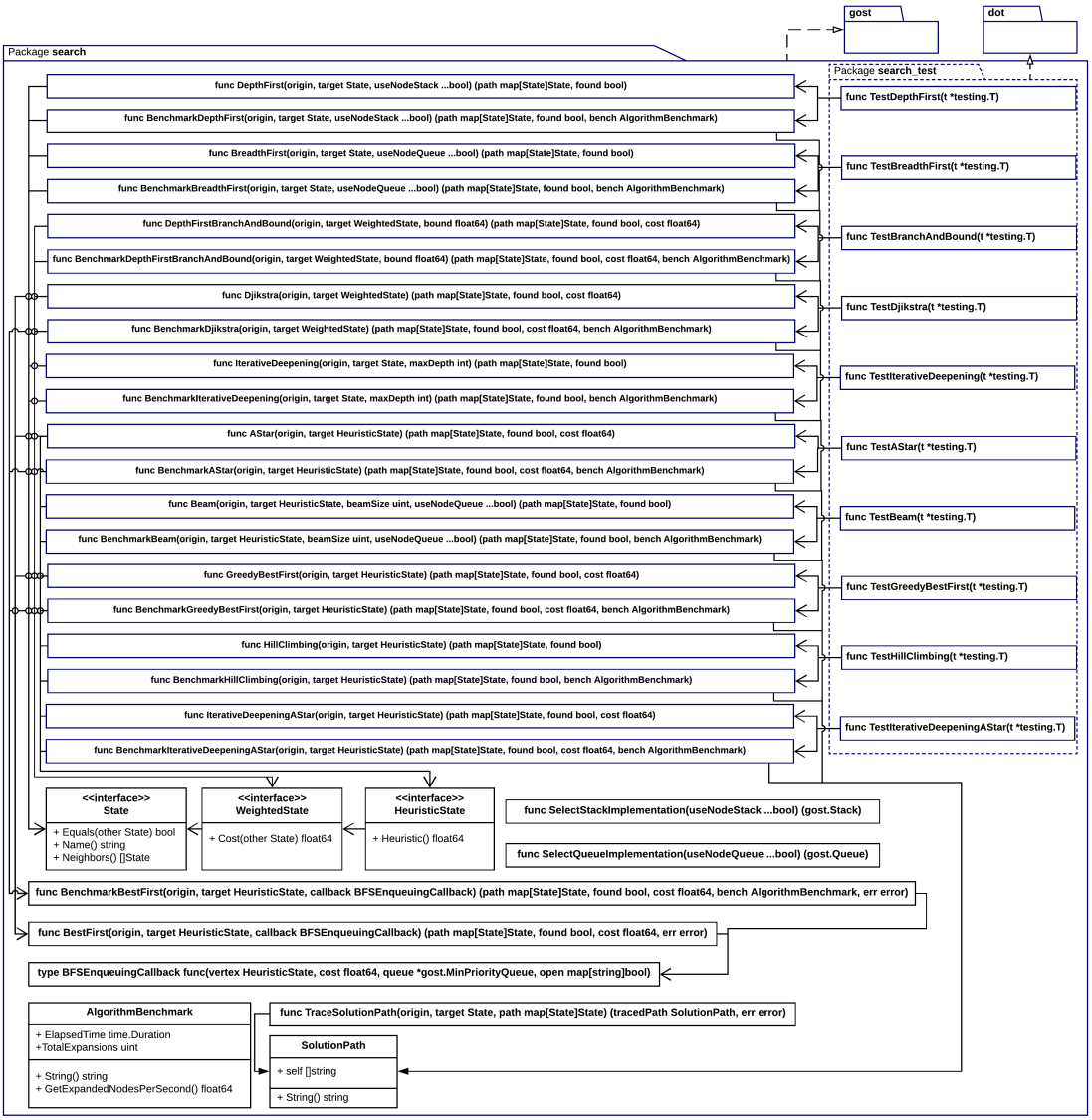


Fig. 2.3. *Type Diagram* for package `search`

2.5. Traceability matrix

Traceability Matrix	TC-01	TC-02	TC-03	TC-04	TC-05	TC-06	TC-07	TC-08
PR-01			×			×	×	×
OR-02	×	×	×	×	×	×	×	×
FR-14				×	×	×	×	×
FR-13				×	×	×	×	×
FR-12				×		×	×	×
FR-11							×	
FR-10					×			
FR-09	×	×			×			
FR-08	×	×						
FR-07					×			
FR-06					×	×	×	×
FR-05					×	×	×	×
FR-04					×	×	×	×
FR-03					×	×	×	×
FR-02	×							
FR-01		×						

Table 2.19. Traceability Matrix

2.6. Planning

Before delving into the implementation details of each of the libraries, a planned roadmap is detailed in the form of a *Gantt* chart. The development of the project itself follows an *agile, test-driven* approach. Despite of this fact, a first usable release of the complete software is represented as a *milestone* but taking into account that the period assigned to development consists of quick iteration cycles over unitary components.

The *Gantt* chart below depicts the project's overall timeline. The three main phases correspond with 1) research and specification phase 2) development and testing phase, and 3) iteration phase, where both latter phases follow an *agile* approach.

Notice how both development and testing phases nearly overlap. This is purposefully presented, as the development is meant to follow a test-driven development, where each unitary block needs its corresponding test, building all the way up to integration tests.

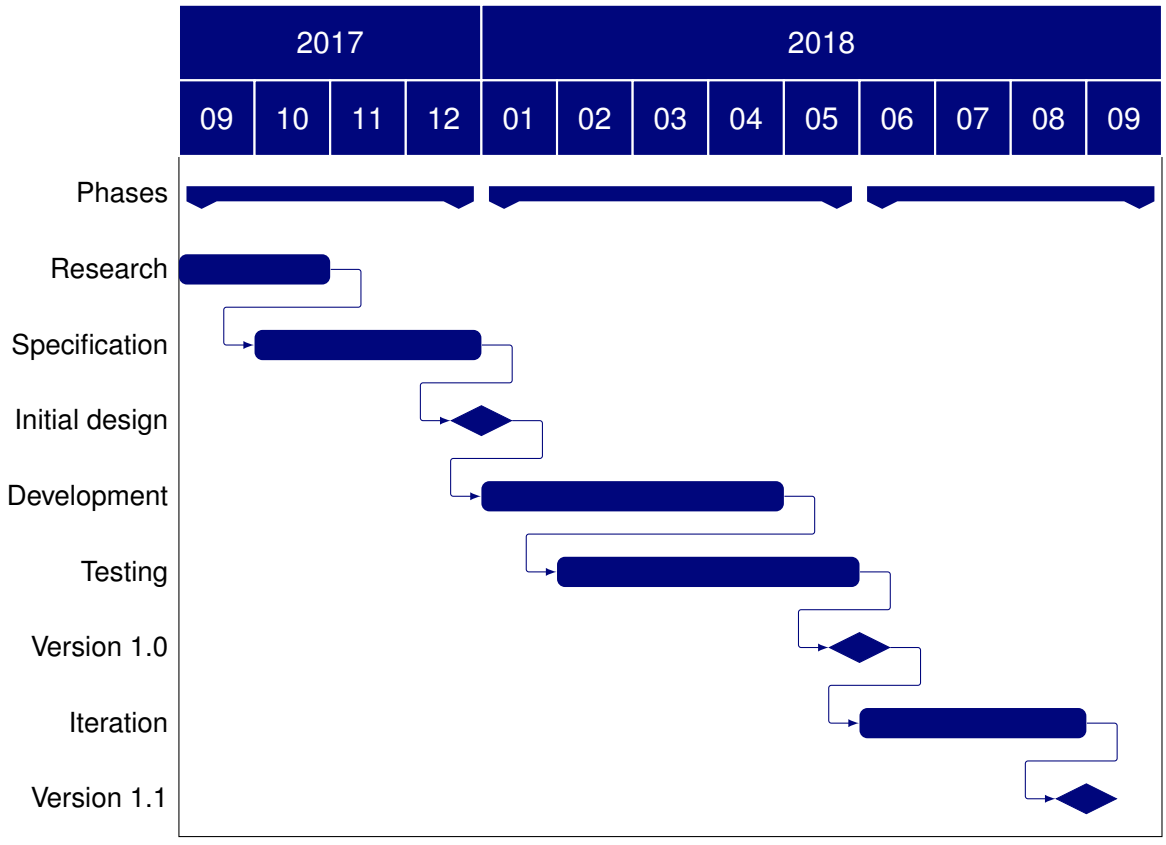


Fig. 2.4. Project Timeline (Gantt chart)

2.7. Management systems

2.7.1. Version Control

The three packages are managed with *git*. Additionally, all of them are publicly hosted on *Github*:

- gost: `https://github.com/christat/gost`
- search: `https://github.com/christat/search`
- dot: `https://github.com/christat/dot`

2.7.2. Testing and Continuous Integration

The project's operational status is kept up-to-date through the usage of *TravisCI*. The aforementioned tool allows re-running the entire test suite on the cloud to ensure no regressions or buggy code remains undetected.

2.7.3. Project management

The task of transforming requirements and design components into tasks and subtasks has been realized employing a physical, analog *Kanban board*. Additionally, time tracking is included by summing up per-task spent time estimates, aggregated in the `Tick` time tracking application (https://www.tickspot.com/time-tracking-app).

3. `gost`: Go Data Structures Library

One of the peculiar aspects of *Go* is the sensibly limited standard library, which omits most common data structures, leaving the programmer with the barebones: `slices` and `maps`. In brief, `slices` are sequential collections of other types. Unlike arrays, slices don't have a fixed size, instead being copied over to new regions of memory every time the container overflows.

The official documentation highly encourages the usage of slices as the language development team has implemented advanced optimizations to ensure a minimal overhead when the inner arrays overflow and require copying, making it a pretty efficient sequential structure.

The other basic data structure, known as `map`, is what is known in classic Computer Science as a *Hash Map*, or a container of `key -> value` pairs where `key` must be unique.

While for many usages those basic structures plus the built-in concurrency of the language may be just about enough to develop complex solutions, search algorithms heavily rely on pre-requisite data structures lacking in the library.

The following table depicts the Data Structures that compose `gost`:

Name	Description
`NodeList`	Single-linked list using a `Node` struct.
`Queue`	Slice based *FIFO* [4] collection.
`NodeQueue`	Single-link list based *FIFO* [4] collection.
`PriorityQueue`	Binary Heap based *FIFO* [4] collection with highest-first priority levels.
`MinPriorityQueue`	Binary Heap based *FIFO* [4] collection with lowest-first priority levels.
`Stack`	slice based *LIFO* [4] collection.
`NodeStack`	Single-linked list based *LIFO* [4] collection.

Table 3.1. `gost` data structures

[4] *FIFO*: First In, First Out; *LIFO*: Last In, First Out.

3.1. `NodeList`: a single-linked sequential container

Even though slices enable us to store lists of data, there are some cases in which it is prefered to avoid the overhead of copying all the items of an array into a new container, specially when the existing array already has a high order of values. For this reason, a Node-based list of data helps us eliminate that overhead: each item in the lists is wrapped into a `Node`.

A `Node` is a structure which holds two fields: the `Data` to be kept in the list, and a `Next`: a pointer to the following item in the list, if any. In this way, copies of all the items into a new container never have to be performed, since the structure operates with references to the next containers.

The entire sequence of Nodes is held together by the `NodeList` struct, which in turn holds three attributes: the `Head` Node, which corresponds with the first item in the list, and the `Tail` Node, which corresponds with the last item in the list, and the `size`, which has to be kept track of to avoid the expensive $\mathcal{O}$(n) traversal.

`NodeList` offers four operators:

- `Append` an item at the end of the list.
- `Retrieve` an item at a specified index.
- `Add` an item at a specified index.
- `Remove` an item at a specified index.

This data structure has the main advantage of eliminating any overhead related to overflowing an underlying container (since there is, in essence, no container, just a sequence of pointers to consecutive elements). The insertion and deletion cost is $\mathcal{O}(1)$, as it only requires to set the new Node's value and pointer, and update the pointer of the previous element[5]. The cost, however, of searching and indexing is $\mathcal{O}$(n) with n being the size of the list, as both require to iterate through all the pointers starting from the head to find the desired element. Both stated complexities can be trivially proven by the reader. More information can also be easily found, for example, in online resources [13].

[5] Provided you have a pointer to the previous element, this aspect is important as the insertion/deletion operations are different from indexing, which is a common misconception when analyzing the complexity of lists

3.2. Queues

A *Queue* is a *FIFO* container of elements. It is the de-facto data structure used to implement a Breadth-First Search algorithm. The `gost` package implements two variants: a slice based one, and a single-linked list based one. In order to ensure the best possible performance for its operations, The theoretical complexity will be briefly addressed and later on the benchmarking procedure and consequent analysis of both variants. Both implementations fulfill the `Queue` interface, which defines the following operations:

- `Enqueue`: Add an item to the last position of the Queue.
- `Dequeue`: Retrieve the first item in the Queue.
- `Size`: Retrieve the number of elements in the Queue.

3.2.1. type `Queue`

The regular `Queue` type uses a `slice` as the container for the elements. The insertion, indexing and removal costs are therefore equivalent to those of *Go*'s built-in `slice`. As slices store their own size (accesed via the `len` operator), the `Size` function is a mere convenience wrapper, required to fulfill the interface signature.

3.2.2. type `NodeQueue`

`NodeQueue` follows the same principle, but using `NodeList` for the underlying container. Unlike its slice-based counterpart, `Size` is an attribute (due to the nature of `NodeList`, length is an attribute that has to be kept track of). The complexities are equivalent to those of `NodeList`, as essentially it's the same structure with a renamed subset of methods.

3.2.3. Benchmarks and comparison

Thanks to *Go*'s powerful integrated benchmarking tool, `gobench`, and the associated helper benchmarking utils, benchmarks for both types of queues (and also stacks!) are easily to implement.

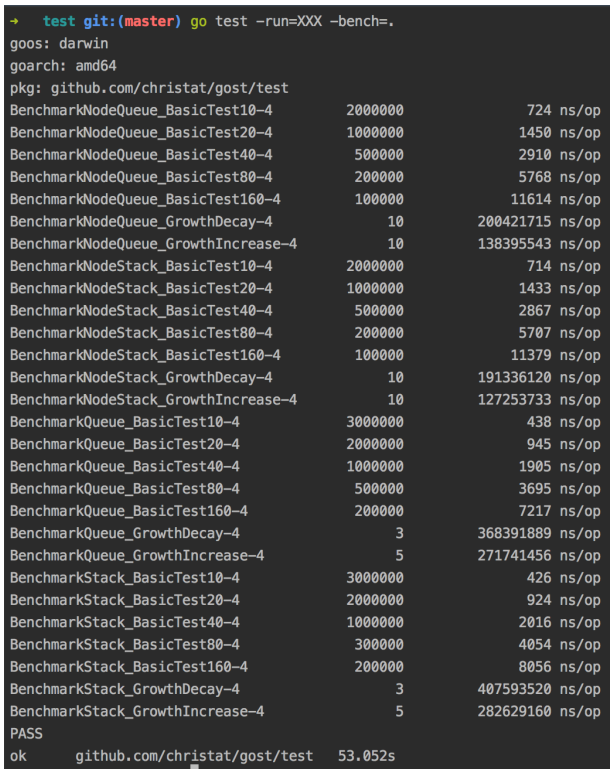


Fig. 3.1. Example of `gobench` output

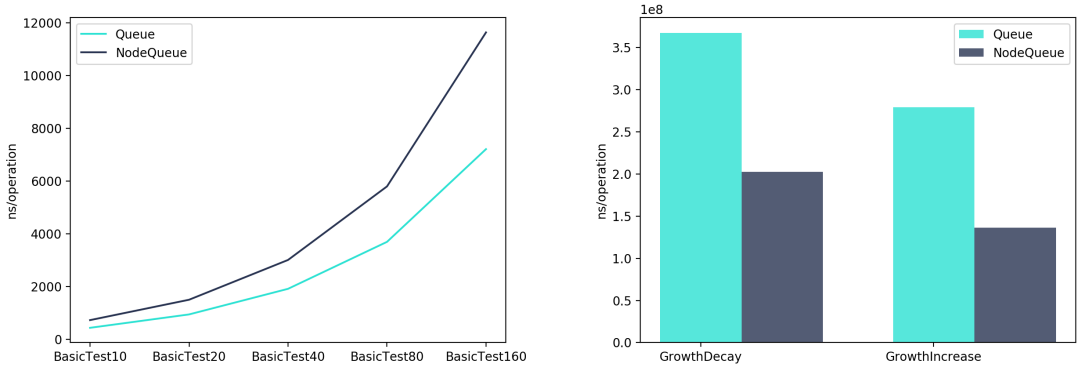
Both implementations (and those of Stacks as well) have been fitted with benchmarking functions which perform an insertion and removal test of 10, 20, 40, 80 and 160 items while the utils compile execution metadata (i.e. nanoseconds taken per operation). The outputs of the functions can be seen in fig. 3.1 with the name `<type>_BasicTest<number>`.

Two additional tests analyze the penalty hit of a growth increase and decay within the data structures; the tests expands and consecutively shrinks the contents by half of the amount of expanded elements, and vice-versa. The tests are set to expand/shrink 1000000 items and perform the inverse operation for half the amount, several times. These tests do not specify the number of attempts; instead, the helper tools themselves decide how many times to run the benchmark. The number of execution runs is provided then as metadata. Said test results can be found in fig. 3.1 with the name `<type>_GrowthDecay` and `<type>_GrowthIncrease`.

The aim of these tests is to measure the performance hit caused by the need to copy a `slice` based queue when full, as well as when it shrinks beyond the minimum *occupancy* threshold, versus the higher cost per operation of using a linked list from the beginning.

A small *Python* script (Appendix A) was employed to process the benchmark data, namely compute the average of 5 benchmark runs for each test. Since `gobench` allows us to run all the benchmark functions in sequence, the data retrieval process consisted in executing it 5 times, pasting the results into a *.csv* file and feeding it into the script.

The results have been plotted using the `matplotlib` library, and displayed below:



(a) Benchmark results (b) Stress test results

Fig. 3.2. Comparison between `Queue` and `NodeQueue`

The benchmark plot 3.2a shows how the `slice`-backed implementation of `Queue` makes the cost per operation lower than that of `NodeQueue`. However, the stress test 3.2b gives a counterpoint: the overhead of copying over an entire slice to a new memory region every time it overflows or gets resized exceeds the cost of using a `NodeQueue`, which doesn't require any sort of container (as it uses random memory addresses to link the nodes together).

This is a revealing (albeit expected) finding that brings a new question: which implementation should be used in the algorithms? The answer to it is also expected - it depends, there's no "silver bullet" to all problems. Hence, the design of the algorithms includes the option to pass a parameter to the function, specifying which implementation should the algoritm use as a flag, defaulting to the `slice`-backed version. The default has been picked considering the intended real-life expected usage: small, didactical examples to learn about Search and Optimization.

3.2.4. type `PriorityQueue` and `MinPriorityQueue`

This particular queue is used in best-first algorithms (in this case, `Djikstra`, `GBFS` and `A*`). Once again, adata structure that has an overall good performance for all of its operands is desirable.

On a higher abstraction level, `PriorityQueue` follows the same principles as a regular Queue, with the addition of item *Priorities*: An item with a higher priority than another ahead of it will ”cut in line” and therefore `Dequeue` sooner.

In practice, neither a common slice nor single-linked list based implementation would give us good results: regardless of which implementation was to be used, each `Dequeue` operation would have an *indexing* cost of O(n), as it would have to find the first item with the highest priority. `Enqueue`-ing would have a cost O(1) and O(n) per implementation, respectively, as explained in section 3.1.

For these reasons, a `heap` structure is employed: a tree-like structure where leaf nodes always have a lower value (in the case of a *maximum heap*) than their parents:

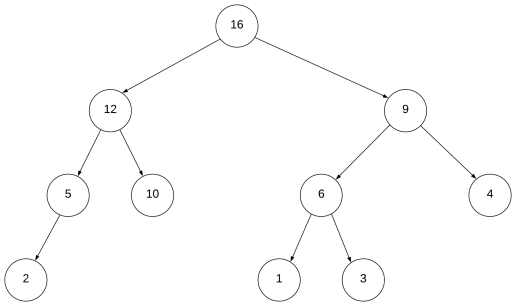


Fig. 3.3. Example of *heap* structure

Go’s standard library includes a `container/heap` interface which can be built upon to have a working `heap`. Both queue operations have a cost of O(log n) [14].

Both implementations differ exclusively in the implementation of the `Less` method defined in the `container/heap` interface: Whereas the regular `PriorityQueue` orders in decreasing order of priority, `MinPriorityQueue` does so in increasing order.

3.3. Stacks

A *Stack* is a *LIFO* container of elements. The implementations must fulfill the `Stack` interface, which defines the following operations:

- `Push`: Add an item to the top of the Stack
- `Pop`: Remove and return the top item in the Stack.
- `Peek`: Retrieve the top item in the Stack, without removing it.
- `Size`: Obtain the number of elements stored in the Stack.

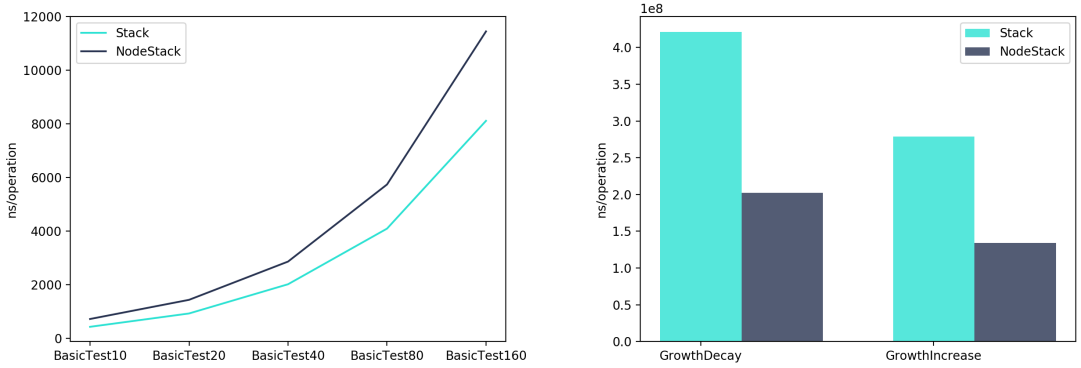
3.3.1. type `Stack`

`slice`-backed implementation: it is essentially a simplified `slice` with renamed methods. As such, it inherits all of `slice`'s operation costs.

3.3.2. type `NodeStack`

`NodeList`-backed implementation. Inherits the operation costs of `NodeList`.

3.3.3. Benchmarks and comparison



(a) Benchmark results

(b) Stress test results

Fig. 3.4. Comparison between `Stack` and `NodeStack`

The pattern shown in the queues comparison is further cemented here: the nanoseconds taken per operation by the `slice`-backed stack is lesser, but offset in the long run by memory region copying costs when the structure is put under stress.

Algorithms requiring the usage of said data structure will hence default to `Queue` for educational purposes, with the configurability to switch them for a `NodeQueue`.

4. `search`: Search Algorithms Library

Before diving into the library itself, I feel obligated to mention the original inspiration behind the thesis, which would be the paper “Una implementación general de los modelos fundamentales de razonamiento inteligente” by Carlos Linares López and Asunción Gómez Pérez [15]. The paper introduces a proposal of a design for a context-independent search algorithms library, with a flexible mechanism for advanced cost and heuristic computations (e.g. multi-objective cost problems), written in C++. Said paper has been used extensively as the reference and basic design and development framework of `search`; however, both designs diverge substantially due in part to the conceptual nature and also the programming languages of choice, respectively: whereas *SAL*, the reference design of the paper, relies heavily on Object-Oriented Programming concepts, including wrappers and helper classes for the generation of statistics, as well as the usage of the *Standard Template Library* of C++ to achieve “generic” implementations, `search` focuses on function composability, as well as a strong interface and type system as the “glue” between the algorithms and the problem space implementations (discussed later). Effectively, `search` can be seen as a set of public functions implementing search algorithms, each of them parameterized to receive problem definitions and return solution data.

The following algorithms are included in `search`:

Function name	Description
`AStar`	A* algorithm.
`Beam`	Beam Search algorith.
`BreadthFirst`	Breadth-First Search algorithm.
`DepthFirst`	Depth-First Search algorithm.
`DepthFirstBranchAndBound`	Depth-First Branch & Bound algorithm.
`Djikstra`	Djikstra algorithm.
`GreedyBestFirst`	Greedy Best First algorithm.
`HillClimbing`	Hill Climbing algorithm.
`IterativeDeepening`	Iterative-Deepening Search algorithm.
`IterativeDeepeningAStar`	Iterative-Deepening A* algorithm.

Table 4.1. `search` algoritm collection

4.1. `interfaces` defined by the package

In order to streamline and unify the set of functions available within the bodies of the algorithms in order to succesfully manipulate the graph and explore the neighbors, compute traversal costs on edges, heuristics, etc. the following Interfaces are defined. Note the gradual composition from one to the next:

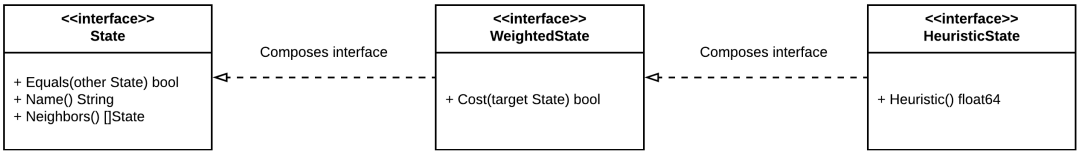


Fig. 4.1. *Type Diagram* of the interfaces exposed by `search`

By defining limited sets of operators, each search algorithm will exclusively have access to the minimum required scope of the implementor type. In the case of the type `Graph` (explained in 5.1), all three interfaces are implemented to enable each algorithm the usage of the required operands. This also adds economy to the implementation as the functions are only required to implement once to cover all three interfaces. Unfortunately, *Go*'s type syntax doesn't allow for greater flexibility; ideally each implementor would expect the function parameters of same interface type to be expected as the same actual underlying type (for instance, making the call to `Neighbors` from a type `Vertex` assume that the returned array items are of type `Vertex`).

4.2. Algorithms Preface

Despite the fact that we have already covered in depth the pre-requisite data structures implemented within package `gost`, it's important to take a page from the famous "No free lunch" theorem, applied to algorithms [16]. This extends to the implementation details of said algorithms; it is simply not feasible to select such a data structure that would perform optimally for any given problem definition.

For that reason, the decision has been carried out to provide alternatives by means of creating interfaces for some of the data structures themselves, and several implementations as described in the previous chapter. The ease of extensibility of said implementations, as long as they fulfill the interface signature, allows to either tweak the algorithms to improve per-case performance, and additionally allows to create tailor made structures for specific needs, whenever required.

Additionally, in order to fulfill the "benchmark manager" requirements that were

originally described in the paper by Linares et al. [15], each algorithm is callable from another function/variant, whose name is prefaced by the 'Benchmark' keyword. Said functions additionally provide as output another type, defined `AlgorithmBenchmark`. The type holds two values: total execution time and total number of node expansions made. Additionally, the type allows obtaining an informative text through the `String` method, and a simple statistic by means of `GetExpandedNodesPerSecond`.

The entirety of the search algorithms library has been engineered with composability and consistency in mind. For that reason, the following conventions apply to any given search function:

- **Input parameters**: Every search algorithm function is provided with a pair of `State` types, both the origin and the desired destination. Any algorithm-specific arguments, such as `beamSize` for Beam Search, the maximum depth in Iterative-Deepening Search, or the initial bound (if desired) in Depth-First Branch and Bound, are passed right after the States. Finally, if the algorithm permits it, a final optional parameter (marked with the `...` syntax) is passed to optionally override the default data structure used internally.
- **Output parameters**: Every search algorithm will at least provide two values: a `path` of the solution (which may be incomplete if not found), and a `found` boolean to indicate whether the algorithm resolved succesfully. Additionally, algorithms which compute costs and heuristics may additionally return the final `cost` of the solution found.
- In the particular case of "Benchmark" algorithm variants, a last output value is provided - the `bench` attribute - an instance of `AlgorithmBenchmark`.

The reader may find it confusing that every algorithm returns a `path` variable, and even more so after inspecting the architectural proposal, realizing that it consists of a `map` of state types. The reasoning behind this data structure choice is that it permits the creation of a utility, called `TraceSolutionPath`, which effectively takes said map, as well as a pair of origin and target states, and recreates the solution path found by the algorithm. The benefit of storing the graph traversal in a map is that some algorithms, such as Djikstra, enable us to find the optimal path to any given node from the source state, hence avoiding the need to recompute optimal paths for several different target states.

4.3. Best-First algorithms helper

Best-First Search can be described as a family of complete algorithms which use two lists of nodes: an open list of nodes awaiting to be expanded, and a closed list of already expanded ones [17]. The most important characteristic is that they are guided by an evaluation function, commonly defined as f(x), which gives a heuristic, or "estimation" of how good is the solution so far. In our case, the Djikstra algoritm, Greedy Best-First and A* are all algorithms which work with the same basic principle, with the only major difference being the evaluation function itself. It is noteworthy that for this family of data structures, a Priority Queue is always employed (at least to store the open list).

For this very reason, the library defines a function that includes all of the possible shared code between said algorithms. The function, called `BestFirst`, takes in a special `callback` parameter, which is a function that receives a set of all the parameters required for each of the aforementioned algorithms to compute their evaluation cost and enqueue themselves in the Priority Queue.

In this way, the actual algorithm functions are simple wrappers around the function implementing the body of Best-First Search, which provide the pre-implemented *callbacks*, along with the user-specified origin and target states.

The following pseudocode adapts the idea from Pearl [17] within the codebase:

```
Data: origin, target, callback
Result: path, found
open = minPriorityQueue.Enqueue(origin)
closed = []
lowestCost = new map()
while open is not empty do
    vertex = open.dequeue()
    closed.add(vertex)
    if vertex equals target then
        found = true
        break
    end
    for neighbor in vertex.Neighbors() do
        if neighbor in closed then
            skip
        end
        cost = callback(vertex, neighbor)
        if cost less than lowestCost[neighbor] then
            lowestCost[neighbor] = cost
            open.add(neighbor)
        end
    end
end
return path, found
```

Fig. 4.2. Pseudocode for Best First helper

4.4. Blind Algorithms

Blind search algorithms, also known as *brute force* search algorithms, compose the family of algorithms which explore the state space without any additional information. Some of them operate directly on the directions/connections between states (which is known as uniform cost search), whereas some other algorithms consider costs when traversing the edges between nodes in the search space graph. The following blind algorithms are provided by `gost`:

4.4.1. func `BreadthFirst`

Breadth-First search is one of the most basic, popular and ubiquitous search algorithms known. It explores all the nodes at a given depth d, until the solution is found or no solution exists at said depth, in which case proceeds to explore all nodes at depth d + 1, etc. Consequently, it will always find a solution which will be optimal, leaving tractability concerns aside, in terms of computational complexity (how long would it take to find the solution node at depth d of a tree with exponential branching factor?). It's actually hard to trace the first research paper employing it, though it's commonly accepted that Edward F. Moore [18] discovered it in the context of maze pathfinding, and Lee [19] re-discovered it in the context of circuit board connections.

The algorithm is implemented iteratively, that is, by means of a `queue` data structure. As such, the input of the algorithm allows to choose among queue-based data structures provided by `gost`. It follows the following pseudocode:

```
Data: origin, target, useNodeQueue
Result: path, found
open = selectQueueImplementation(useNodeQueue)
open.enqueue(origin)
while open is not empty do
    vertex = open.dequeue()
    found = checkVertexAndEnqueueNeighbors(vertex, target, open, path) if
     found then
        break
    end
end
return path, found
```

Fig. 4.3. Pseudocode for BFS algorithm

4.4.2. func `DepthFirst`

Depth-First Search is an antagonistic algorithm to BFS in several ways: it explores trees by depth, choosing at each step a given node's child and not visiting its other neighbors; it isn't complete nor admissible, and it consumes less resources in the case of finding a solution (as it doesn't explore the entire tree it cannot be guaranteed completenes, and less so admissibility). The development of the algorithm has been based on an interesting article proving BFS correctness by using *Isabelle/HOL*, a proof assistant using a formal language [20].

The implementation provided is once again iterative, employing a `Stack`. This has the peculiarity that children are in a different order than a recursive variant, as the nodes are extracted by inverse insertion order in the stack. The following pseudocode describes the implementation used in `gost`:

```
Data: origin, target, useNodeStack
Result: path, found
open = selectStackImplementation(useNodeStack)
open.enqueue(origin)
while open is not empty do
    vertex = open.pop()
    found = checkVertexAndPushNeighbors(vertex, target, open, path) if found
     then
        break
    end
end
return path, found
```

Fig. 4.4. Pseudocode for DFS algorithm

4.4.3. func `DepthFirstBranchAndBound`

Branch and Bound is a family of search algorithms, characterized by iteratively generating a sequence of improving solutions by means of setting a *bound* on the cost of the solution, and a pruning mechanism to discard solutions that don't fit said bound. In this case it applies a depth-first search strategy, with additional cost information to prune unfit candidate solutions.

The implementation relies heavily on source materials from Poole and Mackworth [21]. The following pseudocode excerpt from their materials illustrates the version implemented in `gost`:

```
1: Procedure DFBranchAndBound(G,s,goal,h,bound_0)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         s: start node
5:         goal: Boolean function on nodes
6:         h: heuristic function on nodes
7:         bound_0: initial depth bound (can be ∞ if not specified)
8:     Output
9:         a least-cost path from s to a goal node if there is a solution with cost less than bound_0
10:          or ⊥ if there is no solution with cost less than bound_0
11:    Local
12:        best_path: path or ⊥
13:        bound: non-negative real
14:        Procedure cbsearch(⟨n_0,...,n_k⟩)
15:            if (cost(⟨n_0,...,n_k⟩)+h(n_k) < bound) then
16:            if (goal(n_k)) then
17:                best_path ←⟨n_0,...,n_k⟩
18:                bound ←cost(⟨n_0,...,n_k⟩)
19:            else
20:                for each arc ⟨n_k,n⟩∈A do
21:                    cbsearch(⟨n_0,...,n_k,n⟩)
22:    best_path ←⊥
23:    bound ←bound_0
24:    cbsearch(⟨s⟩)
25:    return best_path
```

Fig. 4.5. DFS Branch And Bound pseudocode from *Artificial Intelligence: Foundations of Computational Agents* [21]

4.4.4. func `Djikstra`

Djikstra is another reputable search algorithm, named after the ACM A.M. Turing-awarded computer scientist Edsger Dijkstra [22]. It belongs to the family of *Best-First* search algorithms. In this case, Djikstra's heuristic function employs only the cost itself (hence f(x) = g(x)), and in fact is a special case of the A* algorithm, being h(x) = 0. Since it explores all nodes and the heuristic never over-estimates the shortest path, it is both complete and admissible. Proof is left for the reader to research.

The implementation of the algoritm uses the previously mentioned `BestFirst` function (4.3), while providing a callback function to compute f(x) and update the underlying Min-Priority Queue.

4.4.5. func `IterativeDeepening`

Iterative Deepening Search is a more advanced algorithm in conceptual terms, based on a simpler algorithm called *Depth-Bound Search*: a depth-first search algorithm that limits the amount of expansions by a certain bound b, representing the depth at which it should stop exploring if a solution has not yet been found.

This simple notion is taken a step further in IDS, by exploring several paths with a limited depth; if no solution is traced, the depth is increased and the paths are re-explored with a higher bound. Even though upon initial impression re-visiting so many nodes may seem taxing, in practice it proves to be a performant solution, compromising between the efficiency of DFS and the completeness of BFS.

The algoritm was originally devised by Richard Korf in 1985 [23]. Both this article and the fantastic chapter on IDS from Poole and Mackworth [21] have been utilized for `search`. Below, the pseudocode extract from the book is provided:

```
1:procedure ID_search(G, s, goal)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         s: start node
5:         goal: Boolean function on nodes
6:     Output
7:         path from s to a node for which goal is true
8:         or ⊥ if there is no such path
9:     Local
10:        hit_depth_bound: Boolean
11:        bound: integer
12:        procedure Depth_bounded_search(⟨n_0, ..., n_k⟩, b)
13:            Inputs
14:                ⟨n_0, ..., n_k⟩: path
15:                b: integer, b ≥ 0
16:            Output
17:                path to goal of length k + b if one exists
18:            if b > 0 then
19:                for each arc ⟨n_k, n⟩ ∈ A do
20:                    res := Depth_bounded_search(⟨n_0, ..., n_k, n⟩, b − 1)
21:                    if res is a path then
22:                        return res
23:            else if goal(n_k) then
24:                return ⟨n_0, ..., n_k⟩
25:            else if n_k has any neighbors then
26:                hit_depth_bound := true
27:    bound := 0
28:    repeat
29:        hit_depth_bound := false
30:        res := Depth_bounded_search(⟨s⟩, bound)
31:        if res is a path then
32:            return res
33:        bound := bound + 1
34:    until not hit_depth_bound
```

Fig. 4.6. IDS pseudocode from *Artificial Intelligence: Foundations of Computational Agents* [21]

4.5. Informed Algorithms

Informed algorithms or *heuristic* search algorithms make use of additional, contextual information on the domain of the problem, formally defined as by means of a *heuristic function* or h(x) which gives a rough "estimate" of the adequacy of the solution so far in the search process. `search` provides the following informed algorithms:

4.5.1. func `AStar`

A* Is famed to be one of the most all-around robust heuristic search algorithms. The algorithm was originally conceived as part of the Shakey project, and defined by Nilsson et al. [24], as an extension of the Dijkstra algorithm. It belongs to the *Best-First* family, and as such is complete; however, admissibility depends on the *heuristic* employed in each problem. The range of *Best-First* algorithms are described further in section 4.3 as they share most of their common logic. In the case of A*, the provided callback function computes f(x) = g(x) + h(x).

An important differentiating factor in the implementation is that due to the nature of the PriorityQueue implementations provided in `gost`, a decrease-key operation is not available and hence it is not possible to implement the algorithm in a typical manner. An alternative approach, as described by Chen et al [25], re-enqueues nodes that would otherwise be decreased in property get queued again, which causes a memory overhead but is offset by the lessened computational cost of a decrease-key operation itself.

4.5.2. func `Beam`

Beam search is a heuristic search algorithm which uses a parameter, the "width" of the beam or b, and enqueues on each iteration the b most promising neighbors. Effectively, Beam behaves like a Breadth-First Search with pruning. As such, it is not complete, and admissibility depends on whether the heuristic function overestimates the shortest path. Beam search was originally presented in the context of scheduling jobs in a flexible manufacturing system by De and Lee [26].

The implementation makes use of a `queue`, which at each step receives a list of at most b neighbors, previously sorted by the value of their heuristic.

4.5.3. func `GreedyBestFirst`

Greedy Best-First search, also known as GBFS or pure heuristic search, is another algorithm within the *Best-First* family. As such, it employs the body of the code in the function provided (4.3) and implements the callback function, which in this case computes f(x) = g(x).

4.5.4. func `HillClimbing`

Hill-Climbing is essentially a corner-case of Beam search, when the value of b = 1. The implementation, however, does not share code with the aforementioned algorithm, due to the specific need of Beam search of sorting children of a given node by the value of h(x). Hill Climbing only keeps the most promising node, permitting a slight simplification of the algorithm itself, when compared to Beam search.

4.5.5. func `IterativeDeepeningAStar`

Iterative-Deepening A* is a heuristic search algorithm that employs IDS's exploration pattern, but applying A* search in each step instead of Depth-First search. Introduced by Richard Korf in 1985 [27], unlike A* it re-expands nodes (as it performs cost-bound searches) but is capable of finding solutions taking linear memory, as it does not keep track of expanded nodes. Additionally, unlike IDS, IDA* explores different depths per branch in the search tree as the bound is not the depth itself, but the value of f(x). The main caveat of IDA* is its incapability to handle state space transpositions.

5. `dot`: *Dot* File Parser

One of the objectives of this project comprises the inclusion of a `package` that understands `dot` language, whose definition is publicly available in the Graphviz specification webpage [1]. The resulting implementation shall provide a `type` that implements the interfaces defined in section 4.1. The implemenation of the parser itself is a rewrite partly based on code by Carlos Linares López of the *libdot* library [28], written in *C++*.

5.1. type `Graph`

The type `Graph` implements the aforementioned interfaces by implementing all the method signatures [1]. As all the information of a graph must be stored within this type, a `struct` is defined with the following attributes (Note that in *Go*, lowercase attributes are considered private.):

- `Name`: stores the name of the graph, as per *dot* spec.
- `CostKey` (optional): stores the key of the attribute to be accessed within the default cost function provided.
- `HeuristicKey` (optional): stores the key of the attribute to be accessed within the default heuristic function provided.
- `CostFunc` (optional): stores a function to override the default cost function. Must comply with the `WeightedState` interface.
- `HeuristicFunc` (optional): stores a function to override the default cost function. Must comply with the `HeuristicState` interface.
- `vertexMap`: maps each unique vertex name to a pointer to said vertex.
- `adjacencyMap`: maps for each unique vertex name a list of adjacent vertices.
- `vertexAttributes`: maps for each unique vertex a map of key-value pairs of attributes.
- `edgeAttributes`: maps for every arc between two vertices (one in each direction if undirected) a map of key-value pairs of attributes.

[1] *Go* interfaces are fulfilled implicitly, hence the lack of keyword `type ... implements <interfaceName>` as in other languages.

Additionally, the following getters, setters and functions are bound to the type:

- `AdjacencyMap`: getter method for the adjacency map.
- `VertexMap`: getter method for the vertex map.
- `GetVertexAttributes`: retrieve all the attributes of a given vertex, in form of a map.
- `SetVertexAttribute`: set an attribute for a given vertex.
- `GetVertexAttribute`: get an attribute by key for a vertex.
- `SetEdgeAttributes`: set the attributes map of an edge.
- `GetEdgeAttributes`: get the attributes map of an edge.
- `SetEdgeAttribute`: set an attribute of an edge.
- `GetEdgeAttribute`: get an attribute by key for an edge.

The idea behind this type is to provide a complete implementation of the interfaces, in such a way that the framework may be used without any extra development, with a relatively complete feature set, such as custom cost and heuristic functions, or access to predefined costs and heuristics within the *dot* file by means of fixed keys.

5.2. CLI Program

Package `dot` comes with its own installable command-line interface (CLI) program. The main purpose is to provide a quick checker for file correctness, as the program will try to construct a `Graph` and will return an error if it fails to do so. The program is configured to accept the following input and optional parameters:

- `-f [path_to_file]` (mandatory): path to file to be parsed.
- `-v` (optional): verbose mode. Prints the chain of tokens detected during parsing.
- `-i` (optional): inspection mode. If the parsing is succesfull, it prints all connections and attributes of the parsed graph.

This is particularly useful for quick graph modeling, so that the syntax can be easily and quickly checked for correctness. Additionally, as `dot` is used as the testing framework for the `search` package, it allows to perform a self-test evaluation before firing any `search` package tests, which otherwise may lead to incorrect results due to regressions unrelated to the search algorithms library.

5.3. Parsing helpers

As in any parser, the mechanism by which the alphabet is broken down is by means of *tokens* - unitary building blocks which compose the language. The building blocks are defined in code by means of regular expressions that "catch" the tokens they represent. A snippet from the source code can be seen in figure 5.1 (below):

Fig. 5.1. Token definitions in *Go*

As parsing the tokens is not enough on its own, each token is wrapped within a corresponding function, or set of functions, to provide meaningful data to the `Graph` instance being constructed. Roughly each of the tokens from the aforemetioned figure have a corresponding `parse` function. All of them use internally the helper function `sliceMatch`, which receives a regular expression and attempts to match the given input to it. If successful, the matched part is consequently "chopped" out of the given input. Some of the regular expression wrapping functions are described below:

- `stripAllComments`: serves as an input pre-processor, as it matches any C/C++ style comments and replaces them by an empty string in the input.
- `parseAttributes`: This function has the peculiarity of looping over the input conditionally; when the token for beginning of attribute section is matched, it attempts to match indefinitely a comma-separated list of attributes, until it finds the token for closing of attribute section. Additionally, the keys and values succesfully parsed are stored in a `map` structure. The values parsed within the loop are additionally attempted a cast into a primitive value, such as `string`, `float64` and `int`.
- `parseVertexAttributes`: a wrapper function for `parseAttributes`, which assigns the parsed map to a specific vertex.
- `parseTargetVertexName`: a function to generate edge connections. Depending on whether it's set in parameters as *directional*, adds one or two entries to the map of adjacencies of our graph.

5.4. Parser

The parser traverses any given `dot` file, scanning for tokens and building up the corresponding `Graph` instance. Figure 5.2 depicts the pseudocode used to parse a file:

Algorithm 1: `dot` file parser and graph instantiator

```
Data: file_stream, verbose
Result: Graph instance with parsed structure and properties.
g = new(Graph)
stripAllComments(file_stream)
parseGraphName(file_stream, g)
parseBlockBegin(file_stream, g)
while not at end of dot block do
    parseVertexName(file_stream, g)
    parseVertexAttributes(file_stream, g)
    parseEdgeType(file_stream, g)
    parseEdgeAttributes(file_stream, g)
    targetVertex = parseVertexName(file_stream, g)
    if targetVertex found then
        g.SetEdgeAttributes(file_stream)
    else
        parseBlockBegin(file_stream, g)
        while block_end not found do
            block_end = parseBlockEnd(file_stream, g)
            if block_end not found then
                parseVertexName(file_stream, g)
            else
                g.SetEdgeAttributes(file_stream)
            end
        end
    end
end
```

Fig. 5.2. Parser pseudocode

The algorithm reads the graph specification from the root, tokenizing the name, stripping comments and parsing every entry within the graph. Said entries are flexible in nature: they may contain vertex and edge attributes, and each entry may point to a single destination vertex or open a new block with a list of destinations. For simplicity, graph manipulation is implicit in the parser functions so that when e.g. an edge type is parsed, the connection is stored in the `Graph`.

6. Applicable regulations

As an open-source search algorithms library, very few regulations may be considered applicable. A brief rundown of the possible regulation vectors is summarized below:

- **Search algorithm patents**: Software patents in general are a delicate matter, with different regulational bodies and laws applying in different regions of the world. A potential risk involved in the development of the search algorithms library is a breach of terms in case of usage of a patented algorithm. Most of the algorithms discussed have been developed in the United States, so as such we can refer to the patent bodies from both U.S. and Europe to inspect whether such a breach has been performed. The *United States Patent and Trademark Office* provides an online service to help searching for existing patents [29]. Legally speaking, *algorithms* as such are not legally patentable as, according to the United States Code, it's only allow to patent "any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof." [30] There is, however, a highly debated legal grey area in which in the past algorithms have been patented by means of a *process* claim, such as Google's algorithm patents [31]. The discussion of the ethical implications of the American legislation is beyond the scope at hand. With regards to European patents, according to the European Patent Convention, Article 52 on Patentable inventions: (1) "European patents shall be granted for any inventions, in all fields of technology, provided that they are new, involve an inventive step and are susceptible of industrial application", (2) "The following in particular shall not be regarded as inventions within the meaning of paragraph 1", (2.a) "schemes, rules and methods for performing mental acts, playing games or doing business, and programs for computers" [32] states that patenting "programs for computers" is not possible, as well as "mathematical methods" and "abstract ideas", under which an algorithm could be filed.

- **Programming language standards**: *Go* is a highly structurized, "boxed" language, with conventions ranging from variable naming to packaging [2]. Thankfully, the task of ensuring compliance with the standard is eased by the automated tools provided, such as `gofmt` - an automated source code formatter, which allows the developer to fix any non-compliant aspects of the developed codebase.

- **Professional responsibility**: The contents of the thesis deliver a complete software solution under the claim of offering a set of working search algorithms. In the hypothetical case in which this software was commercially sold and distributed to third parties, a bilateral contract would sign the exchange of the developement service in exchange for an economical compensation. If the software did not comply with the agreed upon requirements drafted with the client, said client would have the right to take the breached contract to court and file a legal dispute. However, with the software being an open source library of code, such responsibility is reduced exclusively to personal responsibility, which is not in any measure regulable, more so when anyone is allowed to freely distribute any kind of free, open-source software on the internet. This leaves only the ethical matter, which consists of being truthful to the claims made about what is delivered in the software and not hiding any other unsolicited content.

7. Socioeconomic Context & Impact

In a world increasingly interested in AI, Machine Learning related research, the addition of a search algorithms library may sound counterintuitive at first. As mentioned in the introduction, Machine Learning is the hot topic in AI-related research, with an estimation of 57.6 billion US Dollars invested worldwide in Machine Learning by 2021 [4].

However, being an implementation thesis, as compared to an actual research paper, the impact of the presented work cannot be measured with the same metrics as the prior. The magnitude of the impact of the work can however be mesured by means of two key metrics:

- popularity of the framework: while there isn't a direct way to count all usages of the framework, metrics such as number of stars in the repository and forks on *Github* can be used to estimate the impact of the project.
- community adoption: the acceptance of the framework by the developer community can be counted by contributions from other users, be it in form of pull requests, open issues or active maintainers of the code. In the end, it's a similar metric to the above, focused in active collaboration from the community. Many pull requests and open issues might hint at a greater rate of adoption in solutions used in production.

The potential social impact is therefore moderate, assuming a widespread community adoption and usage. With enough support and active maintainers and users, the framework can potentially become a standardized tool used within the research field of Heuristic Search. However, the actual, realistic likelihood of such widespread adoption is rather slim. Active efforts to promote and raise awareness of the framework would have to be made, and a need to actively maintain, debug and improve the framework would arise.

As the project is an open source repository of code, the economic impact can only be established as the quota of potential economic resources saved every time the framework is used, instead of a team shipping their own custom implementation of the algorithms. As such, it's not possible to measure it, mainly due to the impossibility to conduct an exhaustive survey regarding private companies and their products and services, facing obstacles both of scope size and lack of willingness on the corporate side to share internal data.

8. Project Budget

Before concluding the work, the project budget must be evaluated. In development theses such as this one, and per specific University guidelines and requirements, this section attempts to serve as an exercise in economics and business knowledge, stressing the formermost verb.

The table below depicts the total cost of the project:

Concept	Cost
Development costs	**7875 €**
Indirect costs	2400 €
Project cost	**10275 €**
Risk (5%)	513.75 €
Return (12%)	1233 €
Total project cost	**12021.75 €**

Table 8.1. Project associated costs

Direct costs refer to the upfront costs within the project, i.e. the cost of designing and developing a solution to the given problem and any extra infrastructure.

Indirect costs include concepts like utilities, maintenance costs, network and miscleaneous expenditures.

There is additionally an included risk, defined as a margin reserved in case of unsuccesful project stages, such as design or development. In this case, it has been set as a rather low 5% expectation.

The estimated workload of this thesis consists of 25 ects, ranging between 25 and 30 hours each. The total time spent estimation, tracked via app (as defined in 2.7.3) roughly rounds up to 315 hours. Provided a base salary of 25 €/h is set, the cost of engineering makes a total of 7875 €.

The expected profit of the stakeholder is not computable in this case; however, for the sake of completeness a hypothetical return of 12% is provided.

Taxes are not included due to the hypothetical nature of the project.

9. Final Remarks

Throughout these chapters and sections an entire Search Algorithms framework has seen its conception and construction, along with a solid base of Data Structures and a relatively complete parsing solution for `dot` files. Additionally an analysis of the performance of some of the bits and pieces assembled has been provided.

The choice of language has also been an influence in the thesis work itself: for instance, the need to define a purpose-specific diagramming variant of the UML language in Chapter , or the entirety of the `gost` library being a consequence of the language not providing the data structures required, as explained in Chapter .

One extremely convenient aspect of the code design is the possibility to test the algorithms directly using parsed graph examples via the `dot` package - avoiding the need to manually re-implement the required interface for every set of tests in the `search` package, or creating superfluous helper types which fulfill the interface for the sole purpose of testing. Naturally, to enable this, every package search algorithms rely on needs to be thoroughly tested beforehand, which is accomplished via unit and integration tests within the packages themselves.

Overall, the entire idea has proven to be quite a challenging exercise in software engineering: from maximization of code reuse (not repeating simmilar patterns) to the testing, debugging and ensuring that the specification of each of the algorithms is correctly followed.

Hopefully, through the implementation and open-sourcing of this project, a genuine pool of interest is generated and the software proposition can naturally evolve to become a reference search algorithms library, as well as the de-facto educational and experimental tool thanks to the usage of *dot* language.

My personal wish is that somebody finds a nugget of wisdom, practical information or useful code within this project.

9.1. Future Work

Several concerns and/or improvements have been kept out of the project on purpose, mainly due to time and scope restrictions.

One of the biggest and most interesting of such improvements relates to the reinforcing/redesigning the `interface` API for the search algorithms: currently, the user is limited to unparameterized function definitions for both cost functions and heuristic functions. Expanding that flexibility could prove extremely useful. However, due to the rather inflexible and limited type system of *Go*, concepts such as *Union Types* aren't available, and being a compiled + typed language, it's also not possible to resort to the type flexibility of languages such as *TypeScript* or plain *javascript*, where any callback function would be happily accepted by the algorithm as parameter.

A better, more in-depth diagnosis of all available data structures used currently for search and their performance would have also been highly valuable. Unfortunately there's so many possibilities that the work could be expanded into its own research project. A good starting point, however, could be to provide a micro framework by means of interfaces to data structures that can be accepted on a per-algorithm basis. For example, in this way an instantiated data structure could be provided as an argument to the algorithm call, instead of having a selector by means of a boolean variable. This would enable further flexibility to the user if desired. Said implementation must, however, increase caution measures against unexpected runtime events and errors.

Another interesting topic would be expanding the dot parser to accept the full extent of the language, so as to be able to process any entities representable with it.

Bibliography

[1] E. R. Gansner, *The dot language*, 2002. [Online]. Available: https://www.graphviz.org/doc/info/lang.html.

[2] *"about". GoDoc*. [Online]. Available: https://godoc.org/-/about.

[3] S. Pichai, *"a personal google, just for you". Google official blog*, Oct. 2016. [Online]. Available: https://googleblog.blogspot.com/2016/10/a-personal-google-just-for-you.html.

[4] M. Framingham, ""idc spending guide forecasts worldwide spending on cognitive and artificial intelligence systems to reach $57.6 billion in 2021". *IDC*," International Data Corporation, Tech. Rep., Sep. 2017. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS43095417.

[5] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943. DOI: 10.1007/BF02478259. [Online]. Available: https://doi.org/10.1007/BF02478259.

[6] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[7] *"compare search algorithm and deep learning". Google Trends*. [Online]. Available: https://trends.google.com/trends/explore?date=today%205-y&q=search%20algorithm,deep%20learning.

[8] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., Pearson, Ed. Prentice Hall, 2009.

[9] *"tiobe index for go". TIOBE*. [Online]. Available: https://www.tiobe.com/tiobe-index/go/.

[10] *"stack overflow 2018 developer insights. most loved, dreaded, and wanted languages". Stack Overflow*. [Online]. Available: https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted.

[11] *"language conventions". Effective Go*. [Online]. Available: https://golang.org/doc/effective_go.html#names.

[12] O. M. G. Inc., *About the unified modeling language specification version 2.0*. [Online]. Available: https://www.omg.org/spec/UML/2.0/.

[13] *"know thy complexities!" Big O Cheatsheet*. [Online]. Available: http://bigocheatsheet.com/.

[14] J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[15] C. L. López and A. G. Pérez, “Una implementación general de los modelos fundamentales de razonamiento inteligente,” Universidad Politécnica de Madrid, 2001.

[16] D. Wolpert and W. Macready, “No free lunch theorems for search,” Mar. 1996.

[17] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.

[18] E. F. Moore, in *The shortest path through a maze*, In Proceedings of the International Symposium on the Theory of Switching, H. U. Press, Ed., 1959, pp. 285–292.

[19] C. Y. Lee, in *An algorithm for path connection and its applications*, E.-1. IRE Transactions on Electronic Computers, Ed., 1961, pp. 346–365.

[20] T. Nishihara and Y. Minamide, “Depth first search,” *The Archive of Formal Proofs. http://afp. sf. net/entries/Depth-First-Search. shtml*, 2004.

[21] D. Poole and A. K. Mackworth, “Artificial intelligence - foundations of computational agents,” Cambridge University Press, 2010.

[22] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. DOI: 10.1007/BF01386390. [Online]. Available: http://dx.doi.org/10.1007/BF01386390.

[23] R. E. Korf, “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.

[24] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968. DOI: 10.1109/TSSC.1968.300136.

[25] M. Chen, C. R. A., V. Ramachandran, D. Lan Roche, and L. Tong, “Priority queues and dijkstra’s algorithm,” Computer Science Department, University of Texas at Austin, Tech. Rep., 2007.

[26] S. De and A. Lee, “Flexible manufacturing system (fms) scheduling using filtered beam search,” *Journal of Intelligent Manufacturing*, vol. 1, no. 3, pp. 165–183, Sep. 1990. DOI: 10.1007/BF01572636. [Online]. Available: https://doi.org/10.1007/BF01572636.

[27] R. E. Korf, “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985. DOI: https://doi.org/10.1016/0004-3702(85)90084-0.

[28] C. Linares, *"libdot". Atlassian Bitbucket*. [Online]. Available: https://bitbucket.org/clinares/libdot/overview#.

[29] *"search for patents". united states patent and trademark office*. [Online]. Available: https://www.uspto.gov/patents-application-process/search-patents.

[30] *35 u.s.c. §101 (2017)*. [Online]. Available: https://www.gpo.gov/fdsys/pkg/USCODE-2017-title35/pdf/USCODE-2017-title35-partII-chap10-sec101.pdf.

[31] S. Brin, *"information extraction from a database". united states. patent 6,678,681*, Mar. 2000.

[32] *Convention on the grant of european patents, part ii, chapter i, article 52*. [Online]. Available: https://www.epo.org/law-practice/legal-texts/html/epc/2016/e/ar52.html.

Appendix A: Benchmark aggregator python script

```
import csv

class benchmark:
  def __init__(self, id, runs, ns_per_op):
    self.id = id
    self.runs = runs
    self.ns_per_op = ns_per_op

with open('queue_and_stack_benchmarks.csv') as csv_file:
  csv_reader = csv.reader(csv_file, delimiter=',')
  benchmark_ids = set()
  benchmark_runs = []

  for row in csv_reader:
    benchmark_ids.add(row[0])
    benchmark_runs.append(benchmark(row[0], float(row[1]), float(row
        [2])))

  with open('aggregated_benchmarks.csv', 'w') as output_csv:
    writer = csv.writer(output_csv, delimiter=',')

    for id in benchmark_ids:
      total_ns = 0
      total_runs = 0
      counter = 0

      for run in benchmark_runs:
        if run.id == id:
          total_ns += run.ns_per_op
          total_runs += run.runs
          counter += 1

      writer.writerow([id, total_ns / counter, total_runs / counter
          ])
```

Appendix B: packages documentation

The following pages consist of an html export of the online documents, automatically generated with godoc. The reader is encouraged to browse instead through the hyperlinked, online version at:

- https://godoc.org/github.com/christat/gost
- https://godoc.org/github.com/christat/search
- https://godoc.org/github.com/christat/dot

package gost

import "github.com/christat/gost/list"

Package gost is a (minimal) data structures library for Go. Implements several classic data structures such as single-linked lists, stacks and queues (both node and slice based versions).

Index

type Node
type NodeList
- func (list *NodeList) Add(index int, data interface{}) error
- func (list *NodeList) Append(data interface{})
- func (list *NodeList) Remove(index int) (interface{}, error)
- func (list *NodeList) Retrieve(index int) (interface{}, error)
- func (list *NodeList) Size() int

Package Files

node_list.go

type Node

```
type Node struct {
    Data interface{}
    Next *Node
}
```

Basic Node struct, basis of any single-linked list structure.

type NodeList

```
type NodeList struct {
    Head *Node
    Tail *Node
    // contains filtered or unexported fields
}
```

NodeList is a an implementation of a singly linked list. It takes any interface{} and allows:

- Retrieving: obtaining the value contained at any given index within the list.

- Appending: adding a new value at the last position of the list.

- Adding: adding a new value, specifying the index to be inserted at.

- Removing: deleting a value from the list, obtaining it if needed.

Note that the implementation is NOT thread-safe.

func (*NodeList) Add

```
func (list *NodeList) Add(index int, data interface{}) error
```

Add the data passed as parameter at the position designed by index. Returns an error if out of bounds.

func (*NodeList) Append

```
func (list *NodeList) Append(data interface{})
```

Append the data passed as parameter to the end of the list.

func (*NodeList) Remove

```
func (list *NodeList) Remove(index int) (interface{}, error)
```

Remove the item stored at position index in the list. Returns the extracted data or an error if out of bounds.

func (*NodeList) Retrieve

```
func (list *NodeList) Retrieve(index int) (interface{}, error)
```

Retrieve obtains data stored at position index within the list. Returns the data or an error if out of bounds.

func (*NodeList) Size

```
func (list *NodeList) Size() int
```

Size returns the length of the NodeList.

---

package gost

import "github.com/christat/gost/queue"

Index

type MinHeapContents
- func (mhc MinHeapContents) Len() int
- func (mhc MinHeapContents) Less(i, j int) bool
- func (mhc *MinHeapContents) Pop() interface{}
- func (mhc *MinHeapContents) Push(x interface{})
- func (mhc MinHeapContents) Swap(i, j int)

type MinPriorityQueue
- func NewMinPriorityQueue() (pq *MinPriorityQueue)
- func (pq *MinPriorityQueue) Dequeue() interface{}
- func (pq *MinPriorityQueue) Enqueue(item interface{}, priority float64)
- func (pq *MinPriorityQueue) Size() int

type NodeQueue
- func (queue *NodeQueue) Dequeue() interface{}
- func (queue *NodeQueue) Enqueue(data interface{})
- func (queue *NodeQueue) Size() int

type PriorityQueue
- func NewPriorityQueue() (pq *PriorityQueue)
- func (pq *PriorityQueue) Dequeue() interface{}
- func (pq *PriorityQueue) Enqueue(item interface{}, priority float64)
- func (pq *PriorityQueue) Size() int

type Queue

type SliceQueue
- func NewQueue(cap int) *SliceQueue
- func (queue *SliceQueue) Dequeue() interface{}
- func (queue *SliceQueue) Enqueue(data interface{})
- func (queue *SliceQueue) Size() int

Package Files

min_priority_queue.go node_queue.go priority_queue.go queue_interface.go slice_queue.go

type MinHeapContents

```
type MinHeapContents []*priorityItem
```

heapContents implements heap.Interface and holds priorityItems.

func (MinHeapContents) Len

```
func (mhc MinHeapContents) Len() int
```

len returns the length of heapContents.

func (MinHeapContents) Less

```
func (mhc MinHeapContents) Less(i, j int) bool
```

Less responds whether item in index i should be sorted before j (or will take "Less" time to dequeue). If two contents have the same priority, the response will be false as it strictly checks for higher priority.

func (*MinHeapContents) Pop

```
func (mhc *MinHeapContents) Pop() interface{}
```

Dequeue removes the first value to be dequeued from heapContents.

func (*MinHeapContents) Push

```
func (mhc *MinHeapContents) Push(x interface{})
```

Enqueue expects an element x of type *priorityItem and appends it to heapContents.

func (MinHeapContents) Swap

```
func (mhc MinHeapContents) Swap(i, j int)
```

Swap switches places between both priorityItems in the designated indices.

type MinPriorityQueue

```
type MinPriorityQueue struct {
    // contains filtered or unexported fields
}
```

MinPriorityQueue implements a heap-based priority queue, only exposing methods Enqueue() and Dequeue() for simplicity. Inverse priority means that items with lower priority are dequeued faster than higher priority ones. This implementation uses FIFO order as tiebreaker when elements have the same priority.

func NewMinPriorityQueue

```
func NewMinPriorityQueue() (pq *MinPriorityQueue)
```

NewMinPriorityQueue initializes the heap-based priority queue and returns the instance.

func (*MinPriorityQueue) Dequeue

```
func (pq *MinPriorityQueue) Dequeue() interface{}
```

Dequeue removes the item in the MinPriorityQueue with the lowest priority, or insertion order when there's no lower priority contents. If the queue is empty, returns nil.

func (*MinPriorityQueue) Enqueue

```
func (pq *MinPriorityQueue) Enqueue(item interface{}, priority float64)
```

Enqueue adds an interface item and its priority into the MinPriorityQueue.

func (*MinPriorityQueue) Size

```
func (pq *MinPriorityQueue) Size() int
```

Size returns the size of the MinPriorityQueue.

type NodeQueue

```
type NodeQueue struct {
    // contains filtered or unexported fields
}
```

NodeQueue is a single-linked contents backed implementation of queues. It takes any interface{} and allows:

- Enqueuing: inserting an item into the last position of the queue.

- De-queuing: retrieving the first item in the queue.

Note that the implementation is NOT thread-safe.

func (*NodeQueue) Dequeue

```
func (queue *NodeQueue) Dequeue() interface{}
```

Dequeue the head node of the queue. Returns the data or nil if empty.

func (*NodeQueue) Enqueue

```
func (queue *NodeQueue) Enqueue(data interface{})
```

Enqueue a new Node containing data (interface{}) to the tail of the queue.

func (*NodeQueue) Size

```
func (queue *NodeQueue) Size() int
```

Size returns the length of the NodeQueue.

type PriorityQueue

```
type PriorityQueue struct {
    // contains filtered or unexported fields
}
```

PriorityQueue implements a heap-based priority queue, only exposing methods Enqueue() and Dequeue() for simplicity. This implementation uses FIFO order as tiebreaker when elements have the same priority.

func NewPriorityQueue

```
func NewPriorityQueue() (pq *PriorityQueue)
```

NewPriorityQueue initializes the heap-based priority queue and returns the instance.

func (*PriorityQueue) Dequeue

```
func (pq *PriorityQueue) Dequeue() interface{}
```

Dequeue removes the item in the PriorityQueue with the highest priority, or insertion order when there's no higher priority contents. If the queue is empty, returns nil.

func (*PriorityQueue) Enqueue

```
func (pq *PriorityQueue) Enqueue(item interface{}, priority float64)
```

Enqueue adds an interface item and its priority into the PriorityQueue.

func (*PriorityQueue) Size

```
func (pq *PriorityQueue) Size() int
```

Size returns the size of the PriorityQueue.

type Queue

```
type Queue interface {
    Dequeue() interface{}
    Enqueue(data interface{})
    Size() int
}
```

type SliceQueue

```
type SliceQueue struct {
    // contains filtered or unexported fields
}
```

SliceQueue is a slice-backed implementation of queues. It takes any type implementing interface{} and allows:

- Enqueuing: inserting an item into the last position of the queue.

- De-queuing: retrieving the first item in the queue.

Note that the implementation is NOT thread-safe.

func NewQueue

```
func NewQueue(cap int) *SliceQueue
```

NewQueue creates a new queue with initial len() zero and capacity cap.

func (*SliceQueue) Dequeue

```
func (queue *SliceQueue) Dequeue() interface{}
```

Dequeue the head node of the queue. Returns the data or nil if empty.

func (*SliceQueue) Enqueue

```
func (queue *SliceQueue) Enqueue(data interface{})
```

Enqueue a new node containing data (interface{}) to the tail of the queue.

func (*SliceQueue) Size

```
func (queue *SliceQueue) Size() int
```

Size returns the length of the queue's underlying slice.

---

package gost

import "github.com/christat/gost/stack"

Index

type NodeStack
- func (stack *NodeStack) Peek() interface{}
- func (stack *NodeStack) Pop() interface{}
- func (stack *NodeStack) Push(data interface{})
- func (stack *NodeStack) Size() int

type SliceStack
- func NewStack(cap int) *SliceStack
- func (stack *SliceStack) Peek() interface{}
- func (stack *SliceStack) Pop() interface{}
- func (stack *SliceStack) Push(data interface{})
- func (stack *SliceStack) Size() int

type Stack

Package Files

node_stack.go slice_stack.go stack_interface.go

type NodeStack

```
type NodeStack struct {
    // contains filtered or unexported fields
}
```

NodeStack is a single-linked list backed implementation of stacks. It takes any interface{} and allows:

- Pushing: adding a new element on top of the stack.

- Popping: retrieving the element on top of the stack.

- Peeking: obtaining the element on top of the stack without removing it.

Note that the implementation is NOT thread-safe.

func (*NodeStack) Peek

```
func (stack *NodeStack) Peek() interface{}
```

Peek at the content of the stack head (nil if empty) without removing it afterwards.

func (*NodeStack) Pop

```
func (stack *NodeStack) Pop() interface{}
```

Dequeue the head node from the stack. Returns the data or nil if empty.

func (*NodeStack) Push

```
func (stack *NodeStack) Push(data interface{})
```

Enqueue a new node containing data (interface{}) into the stack.

func (*NodeStack) Size

```
func (stack *NodeStack) Size() int
```

Size returns the depth of the current NodeStack.

type SliceStack

```
type SliceStack struct {
    // contains filtered or unexported fields
}
```

SliceStack is a slice-backed implementation of stacks. It takes any type implementing interface{} and allows:

- Pushing: adding a new element on top of the stack.

- Popping: retrieving the element on top of the stack.

- Peeking: obtaining the element on top of the stack without removing it.

Note that the implementation is NOT thread-safe.

func NewStack

```
func NewStack(cap int) *SliceStack
```

NewStack creates a new stack with initial len() zero and capacity cap.

func (*SliceStack) Peek

```
func (stack *SliceStack) Peek() interface{}
```

Peek at the content of the stack Head (nil if empty) without removing it afterwards.

func (*SliceStack) Pop

```
func (stack *SliceStack) Pop() interface{}
```

Dequeue the head node from the stack. Returns the data or nil if empty.

func (*SliceStack) Push

```
func (stack *SliceStack) Push(data interface{})
```

Enqueue a new node containing data of type interface{} into the stack.

func (*SliceStack) Size

```
func (stack *SliceStack) Size() int
```

Size returns the length of the stack's underlying slice.

type Stack

```
type Stack interface {
    Peek() interface{}
    Pop() interface{}
    Push(data interface{})
    Size() int
}
```

---

package dot

import "github.com/christat/dot"

Package dot provides a .dot parser implementation and a graph type

Index

func BuildGraph(g *Graph, vertexMap map[string]*Vertex, adjacencyMap map[string][]search.State, vertexAttributes map[string]map[string]interface{}, edgeAttributes map[string]map[string]map[string]interface{})
func Parse(fileStream []byte, verboseFlag bool) (bool, *Graph)
func ParseFile(filePath string, verbose ...bool) (bool, *Graph)
type Graph
- func NewGraph() (g *Graph)
- func (g *Graph) AdjacencyMap() map[string][]search.State
- func (g *Graph) GetEdgeAttribute(origin string, target string, attribute string) (interface{}, error)
- func (g *Graph) GetEdgeAttributes(origin string, target string) (map[string]interface{}, error)
- func (g *Graph) GetVertexAttribute(vertex string, attribute string) (value interface{}, err error)
- func (g *Graph) GetVertexAttributes(vertex string) (value map[string]interface{}, err error)
- func (g *Graph) SetEdgeAttribute(origin string, target string, isUndirected bool, attribute string, value interface{})
- func (g *Graph) SetEdgeAttributes(origin string, target string, isDirectional bool, edgeAttributes map[string]interface{})
- func (g *Graph) SetVertexAttribute(vertex string, attribute string, value interface{})
- func (g *Graph) VertexMap() map[string]*Vertex

type Vertex
- func NewVertex(name string, graph *Graph) *Vertex
- func (v *Vertex) Cost(target search.State) float64
- func (v *Vertex) Equals(other search.State) bool
- func (v *Vertex) Heuristic() float64
- func (v *Vertex) Name() string
- func (v *Vertex) Neighbors() (neighbors []search.State)

Package Files

graph.go parser.go parser_expressions.go parser_utils.go vertex.go

func BuildGraph

```
func BuildGraph(
    g *Graph,
    vertexMap map[string]*Vertex,
    adjacencyMap map[string][]search.State,
    vertexAttributes map[string]map[string]interface{},
    edgeAttributes map[string]map[string]map[string]interface{})
```

Build graph from existing data

func Parse

```
func Parse(fileStream []byte, verboseFlag bool) (bool, *Graph)
```

Parse parses the fileStream, building a Graph instance or returning false otherwise.

func ParseFile

```
func ParseFile(filePath string, verbose ...bool) (bool, *Graph)
```

ParseFile wraps the Parse() function with a file reader to get a fileStream ([]byte) if the file exists. Returns a pointer to a Graph instance or false if reading the file or parsing failed.

type Graph

```
type Graph struct {
    Name          string
    Type          string
    CostKey       string
    HeuristicKey  string
    CostFunc      func(origin, target *Vertex) float64
    HeuristicFunc func(vertex *Vertex) float64
    // contains filtered or unexported fields
}
```

Graph contains the topology and attributes of a Graph, including name, type, and adjacency map and vertex/edge attributes. Additionally, it is the backbone of the Vertex type, which implements the interface search.State from github.com/christat/search. This means we can use Graph to perform search with the algorithms provided in the aforementioned library.

func NewGraph

```
func NewGraph() (g *Graph)
```

NewGraph creates and returns a pointer to a new Graph.

func (*Graph) AdjacencyMap

```
func (g *Graph) AdjacencyMap() map[string][]search.State
```

AdjacencyMap returns the adjacency map of the graph.

func (*Graph) GetEdgeAttribute

```
func (g *Graph) GetEdgeAttribute(origin string, target string, attribute string) (interface{}, error)
```

GetEdgeAttribute obtains the desired attribute of an edge (defined by the vertices origin -> target). If the edge is undirected it is assumed that the map will hold the same properties in both directions, making one fetch enough.

func (*Graph) GetEdgeAttributes

```
func (g *Graph) GetEdgeAttributes(origin string, target string) (map[string]interface{}, error)
```

GetEdgeAttributes obtains all the attributes of the edge (defined by the vertices origin -> target). If the edge is undirected it is assumed that the map will hold the same properties in both directions, making one fetch enough.

func (*Graph) GetVertexAttribute

```
func (g *Graph) GetVertexAttribute(vertex string, attribute string) (value interface{}, err error)
```

GetVertexAttributes obtains the desired attribute of vertex. If not found, an error value is returned instead

func (*Graph) GetVertexAttributes

```
func (g *Graph) GetVertexAttributes(vertex string) (value map[string]interface{}, err error)
```

GetVertexAttributes allows obtaining the map of attributes for a given vertex.

func (*Graph) SetEdgeAttribute

```
func (g *Graph) SetEdgeAttribute(origin string, target string, isUndirected bool, attribute string, value interface{})
```

SetEdgeAttribute adds the desired attribute to an edge (defined by the vertices origin -> target)

If isUndirected is true, the property is set for both directions of the edge.

func (*Graph) SetEdgeAttributes

```
func (g *Graph) SetEdgeAttributes(origin string, target string, isDirectional bool, edgeAttributes map[string]interface{})
```

SetEdgeAttributes provides an easy way to set a map of attributes for a specific edge (defined by the vertices origin -> target). if isDirectional is false, the same property will be set in both origin -> target and target -> origin.

func (*Graph) SetVertexAttribute

```
func (g *Graph) SetVertexAttribute(vertex string, attribute string, value interface{})
```

SetVertexAttribute allows to add an attribute to an existing map of attributes for a given vertex.

func (*Graph) VertexMap

```
func (g *Graph) VertexMap() map[string]*Vertex
```

VertexMap returns a map linking vertex names to their instances.

type Vertex

```
type Vertex struct {
    // contains filtered or unexported fields
}
```

func NewVertex

```
func NewVertex(name string, graph *Graph) *Vertex
```

New vertex allows to easily generate a vertex, providing the underlying graph instance and its unique name.

func (*Vertex) Cost

```
func (v *Vertex) Cost(target search.State) float64
```

Cost relies on the underlying graph structure to obtain either a cost function to traverse from v to target, or alternatively a cost key if the cost is coded into the graph description. Alternatively, it returns a default cost of 10e9 as a measure of caution.

func (*Vertex) Equals

```
func (v *Vertex) Equals(other search.State) bool
```

Equals implements the search.State interface, comparing two instances of a Vertex by name (by dot standards, they should be unique).

func (*Vertex) Heuristic

```
func (v *Vertex) Heuristic() float64
```

Heuristic, similarly to the Cost method, relies on either a function or a key passed as an attribute of the underlying graph. As a fallback, a heuristic value of 0 is returned.

func (*Vertex) Name

```
func (v *Vertex) Name() string
```

Name returns the unique identifier of the Vertex, i.e. its name.

func (*Vertex) Neighbors

```
func (v *Vertex) Neighbors() (neighbors []search.State)
```

Neighbors allows to obtain a map of adjacent vertices to the caller.

Directories

Path	Synopsis
cli	

Package dot imports 7 packages (graph) and is imported by 1 packages. Updated about a month ago. Refresh now. Tools for package owners.

package cli

import "github.com/christat/dot/cli"

Index

Package Files

main.go

Package cli imports 4 packages (graph). Updated about a month ago. Refresh now. Tools for package owners.

package search

import "github.com/christat/search"

Index

func BenchmarkBestFirst(origin, target HeuristicState, callback BFSEnqueuingCallback) (path map[State]State, found bool, cost float64, bench AlgorithmBenchmark, err error)
func BestFirst(origin, target HeuristicState, callback BFSEnqueuingCallback) (path map[State]State, found bool, cost float64, err error)
func SelectQueueImplementation(useNodeQueue ...bool) gost.Queue
func SelectStackImplementation(useNodeStack ...bool) stack.Stack
type AlgorithmBenchmark
- func (ab AlgorithmBenchmark) GetExpandedNodesPerSecond() float64
- func (ab AlgorithmBenchmark) String() string

type BFSEnqueuingCallback
type HeuristicState
type SolutionPath
- func TraceSolutionPath(origin, target State, path map[State]State) (tracedPath SolutionPath, err error)
- func (s SolutionPath) String() string

type State
type WeightedState

Package Files

algorithm_utils.go benchmark_types.go best_first_helper.go solution_tracer.go state_types.go

func BenchmarkBestFirst

```
func BenchmarkBestFirst(origin, target HeuristicState, callback BFSEnqueuingCallback) (path map[State]State, found bool, cost float64, bench AlgorithmBenchmark, err error)
```

func BestFirst

```
func BestFirst(origin, target HeuristicState, callback BFSEnqueuingCallback) (path map[State]State, found bool, cost float64, err error)
```

Best First Search underpins several algorithms, such as Greedy BFS or A*. The main difference comes in the enqueuing logic, which is specific to the algorithm itself.

func SelectQueueImplementation

```
func SelectQueueImplementation(useNodeQueue ...bool) gost.Queue
```

SelectQueueImplementation is a shared util to return a data structure implementing the gost.Queue interface. By default, the slice-backed queue is used. When useNodeQueue is set to true, a single linked list variant is returned.

func SelectStackImplementation

```
func SelectStackImplementation(useNodeStack ...bool) stack.Stack
```

SelectStackImplementation is a shared util to return a data structure implementing the gost.Stack interface. By default, the slice-backed stack is used. When useNodeStack is set to true, a single linked list variant is returned.

type AlgorithmBenchmark

```
type AlgorithmBenchmark struct {
    ElapsedTime     time.Duration
    TotalExpansions uint
}
```

Convenience wrapper to obtain extra information regarding the search performed with Benchmark_<Algorithm> variants.

func (AlgorithmBenchmark) GetExpandedNodesPerSecond

```
func (ab AlgorithmBenchmark) GetExpandedNodesPerSecond() float64
```

func (AlgorithmBenchmark) String

```
func (ab AlgorithmBenchmark) String() string
```

type BFSEnqueuingCallback

```
type BFSEnqueuingCallback func(vertex HeuristicState, cost float64, queue *gost.MinPriorityQueue, open map[string]bo
```

Each algorithm decides how to enqueue its nodes. The callback should provide any neccesary parameters.

type HeuristicState

```
type HeuristicState interface {
    Heuristic() float64
    WeightedState
}
```

HeuristicState composes WeightedState, requiring additionally a Heuristic() function.

type SolutionPath

```
type SolutionPath []string
```

SolutionPath is a convenience wrapper for []string to print a neat origin-to-target path string.

func TraceSolutionPath

```
func TraceSolutionPath(origin, target State, path map[State]State) (tracedPath SolutionPath, err error)
```

TraceSolutionPath allows to invert/interpret the result given by a given search algorithm as a slice of state names.

func (SolutionPath) String

```
func (s SolutionPath) String() string
```

type State

```
type State interface {
    Equals(other State) bool
    Name() string
    Neighbors() []State
}
```

The State interface models a specific state in the state space of a problem. We need as bare minimum two functions:

```
- Asserting if a State Equals() another
- Obtaining the Neighbors() of this State
```

type WeightedState

```
type WeightedState interface {
    Cost(target State) float64
    State
}
```

HeuristicState composes a regular State, requiring additionally an inter-state Cost() function.

Directories

Path	Synopsis
blind	
informed	

Package search imports 5 packages (graph) and is imported by 3 packages. Updated about a month ago. Refresh now. Tools for package owners.

package search

import "github.com/christat/search/blind"

Index

func BenchmarkBreadthFirst(origin, target search.State, useNodeQueue ...bool) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
func BenchmarkDepthFirst(origin, target search.State, useNodeStack ...bool) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
func BenchmarkDepthFirstBranchAndBound(origin, target search.WeightedState, bound float64) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
func BenchmarkDjikstra(origin, target search.WeightedState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
func BenchmarkIterativeDeepening(origin, target search.State, maxDepth int) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
func BreadthFirst(origin, target search.State, useNodeQueue ...bool) (path map[search.State]search.State, found bool)
func DepthFirst(origin, target search.State, useNodeStack ...bool) (path map[search.State]search.State, found bool)
func DepthFirstBranchAndBound(origin, target search.WeightedState, bound float64) (path map[search.State]search.State, found bool, cost float64)
func Djikstra(origin, target search.WeightedState) (path map[search.State]search.State, found bool, cost float64)
func IterativeDeepening(origin, target search.State, maxDepth int) (path map[search.State]search.State, found bool)

Package Files

breadth_first.go depth_first.go depth_first_branch_and_bound.go djikstra.go iterative_deepening.go utils.go

func BenchmarkBreadthFirst

```
func BenchmarkBreadthFirst(origin, target search.State, useNodeQueue ...bool) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
```

Benchmark variant of BreadthFirst. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkDepthFirst

```
func BenchmarkDepthFirst(origin, target search.State, useNodeStack ...bool) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
```

Benchmark variant of DepthFirst. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkDepthFirstBranchAndBound

```
func BenchmarkDepthFirstBranchAndBound(origin, target search.WeightedState, bound float64) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
```

Benchmark variant of DepthFirstBranchAndBound. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkDjikstra

```
func BenchmarkDjikstra(origin, target search.WeightedState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
```

Benchmark variant of Djikstra. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkIterativeDeepening

```
func BenchmarkIterativeDeepening(origin, target search.State, maxDepth int) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
```

Benchmark variant of IterativeDeepening. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BreadthFirst

```
func BreadthFirst(origin, target search.State, useNodeQueue ...bool) (path map[search.State]search.State, found bool)
```

BreadthFirst implements the Breadth First Search algorithm. The origin and target states must fulfill the State interface. Optionally, a single-linked list backed stack can be enforced with useNodeQueue.

func DepthFirst

```
func DepthFirst(origin, target search.State, useNodeStack ...bool) (path map[search.State]search.State, found bool)
```

DepthFirst implements the Depth First Search algorithm. The origin and target states must fulfill the State interface. Optionally, a single-linked list backed stack can be enforced with useNodeStack.

func DepthFirstBranchAndBound

```
func DepthFirstBranchAndBound(origin, target search.WeightedState, bound float64) (path map[search.State]search.State, found bool, cost float64)
```

BranchAndBound performs depth search iteratively. An upper bound is set every time a solution is found, pruning costlier descendants and stopping once no better solution was found. Because of its nature (minimization of positive costs) it is not expected to work correctly with negative costs. Maximization problems should be redefined accordingly. Paramter bound can be left as default float64 (0); the algorithm will assume an initial bound of plus infinity.

func Djikstra

```
func Djikstra(origin, target search.WeightedState) (path map[search.State]search.State, found bool, cost float64)
```

Djikstra implements the well known algorithm of said name. Even though the function returns the shortest path between two vertices in a graph, an optimal traversal between any two points can be built by using the return path map and func TraceSolutionPath. If a path exists, it will be indicated by found and the traversal cost returned.

func IterativeDeepening

```
func IterativeDeepening(origin, target search.State, maxDepth int) (path map[search.State]search.State, found bool)
```

IterativeDeepening implements recursive IDS. It will look for optimal solutions reaching target from origin. The depth bound is slowly increased until reaching maxDepth.

---

package search

import "github.com/christat/search/informed"

Index

func AStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64)
func Beam(origin, target search.HeuristicState, beamSize uint, useNodeQueue ...bool) (path map[search.State]search.State, found bool)
func BenchmarkAStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
func BenchmarkBeam(origin, target search.HeuristicState, beamSize uint, useNodeQueue ...bool) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
func BenchmarkGreedyBestFirst(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
func BenchmarkHillClimbing(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
func BenchmarkIterativeDeepeningAStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
func GreedyBestFirst(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64)
func HillClimbing(origin, target search.HeuristicState) (path map[search.State]search.State, found bool)
func IterativeDeepeningAStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64)

Package Files

a_star.go beam.go greedy_best_first.go hill_climbing.go iterative_deepening_a_star.go

func AStar

```
func AStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64)
```

AStar implements the A* algorithm. Even though the function returns the shortest path between two vertices in a graph, an optimal traversal between any two points can be built by using the return path map and func TraceSolutionPath. If a path exists, it will be indicated by found and the traversal cost returned.

func Beam

```
func Beam(origin, target search.HeuristicState, beamSize uint, useNodeQueue ...bool) (path map[search.State]search.State, found bool)
```

Beam implements Beam Search. On each execution step, the most promising set of descendants (i.e. with the lowest Heuristic value) are enqueued, discarding the others (hence not keeping them in the queue). In practice, Beam Search behaves like a pruning-enabled Breadth-First Search, retaining at each expansion a maximum of descendants marked by beamSize.

func BenchmarkAStar

```
func BenchmarkAStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
```

Benchmark variant of AStar. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkBeam

```
func BenchmarkBeam(origin, target search.HeuristicState, beamSize uint, useNodeQueue ...bool) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
```

Benchmark variant of Beam. It measures execution parameters (time, nodes expanded) them in a

search.AlgorithmBenchmark entity.

func BenchmarkGreedyBestFirst

```
func BenchmarkGreedyBestFirst(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
```

Benchmark variant of AStar. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkHillClimbing

```
func BenchmarkHillClimbing(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, bench search.AlgorithmBenchmark)
```

Benchmark variant of HillClimbing. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func BenchmarkIterativeDeepeningAStar

```
func BenchmarkIterativeDeepeningAStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64, bench search.AlgorithmBenchmark)
```

Benchmark variant of IterativeDeepeningAStar. It measures execution parameters (time, nodes expanded) them in a search.AlgorithmBenchmark entity.

func GreedyBestFirst

```
func GreedyBestFirst(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64)
```

AStar implements the A* algorithm. Even though the function returns the shortest path between two vertices in a graph, an optimal traversal between any two points can be built by using the return path map and func TraceSolutionPath. If a path exists, it will be indicated by found and the traversal cost returned.

func HillClimbing

```
func HillClimbing(origin, target search.HeuristicState) (path map[search.State]search.State, found bool)
```

HillClimbing implements a heuristic-based Hill Climbing algorithm. On each execution step, the most promising descendant (i.e. with the lowest Heuristic value) is further expanded, discarding the others (hence not keeping them in any collection).

func IterativeDeepeningAStar

```
func IterativeDeepeningAStar(origin, target search.HeuristicState) (path map[search.State]search.State, found bool, cost float64)
```

IterativeDeepeningAStar implements the IDA* algorithm. It performs a series of depth searches bounded by the minimum f value (cost + heuristic). if no solution is found, the bound is updated with the minimum explored f value to continue the depth search.
